Reusing Work and Avoiding Redundant Work in Propositional Logic Solvers

by

Randy Hickey

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

Department of Computer Science
University of Toronto

Reusing Work and Avoiding Redundant Work in Propositional Logic Solvers

Randy Hickey
Doctor of Philosophy

Department of Computer Science
University of Toronto
2024

Abstract

Boolean satisfiability (SAT) is a fundamental and well-studied problem in computer science. Many practical applications in industry and in other areas of computer science rely on the speed of SAT solvers and other related propositional logic solvers. When running SAT solvers, one can often observe a lot of repeated computations and redundancy during the solvers' execution. Although there have been many ideas to address this redundancy and make SAT solvers more effective throughout the years, there still remains a lot of redundant work that can be easily observed and identified in state-of-the-art solvers, such as repeated propagation steps, finding cores extremely similar to one another, etc. In this thesis, we set out to identify some of this redundant work and then propose methods to eliminate it. Aside from proving our techniques work in theory, we also demonstrate their effectiveness empirically, by implementing them in multiple state-of-the-art solvers and improving their performance on a wide set of benchmarks that are standard in the field.

The first technique addresses assumption-based SAT, and more specifically the redundancy in the unit propagation process of the assumptions. Our technique was to make a simple change from the standard way of unit propagating assumptions and unit propagate them all at once. This led to a dramatic improvement in the performance of mutliple complete MaxSAT solvers.

The next main technique addresses the redundancy that occurs every time a SAT solver backtracks and continues its redescent down the search tree, where it will reconstruct most of the same assignment trail again. In order to address this, we introduced a simple technique to store the assignment trail separately and used it to re-construct the trail without doing all of the unit propagation steps that otherwise would have occurred. This improved multiple state-of-the-art SAT solvers.

Our last technique addresses the redundancy that occurs during anytime MaxSAT solving, where it turns out what incomplete MaxSAT solvers often do is find multiple MaxSAT solutions consecutively that are extremely similar to one another. By using the Large Neighbourhood Search (LNS) framework, we developed an algorithm for anytime MaxSAT which sets up the MaxSAT problem as an instance of incremental MaxSAT and searches an entire neighbourhood of solutions exhaustively, while removing some redundant computations. This technique managed to improve the performance of the best performing anytime MaxSAT solvers.

Acknowledgements

First, I would like to thank my PhD advisor, Fahiem Bacchus. Back when I was first choosing which grad school to attend, I specifically sought to study under Fahiem's direction. I consider myself very lucky to have gotten a chance to do that. Fahiem taught me so much about SAT solving, taught me how to visualize complex algorithms, taught me how to write rigorous and compelling papers, and most importantly taught me how to do proper science. Fahiem was involved as a collaborator in every step of this thesis, from generating all of the ideas to writing all of the papers to implementing all of the techniques in programming code. Beyond that, his patience, unwavering support, and ability to make research fun made my graduate education so much more enjoyable. Fahiem has left an immeasurably positive impact on my life which I will never forget.

Next, I would like to thank Sheila McIlraith and Eldan Cohen for agreeing to become my co-supervisors under very difficult circumstances. Their encouragement and insistence on getting me to the finish line was essential in getting this thesis completed. This thesis would not be as well written or as well organized without their help. Sheila was part of my supervisory committee from the very beginning and always provided a great wealth of knowledge and context about the broader field in general. She encouraged me to think about the bigger picture, and how my research might be more widely applicable. Eldan joined my committee later, but immediately contributed a fresh perspective. He is always able to recognize when our research progress is stagnating and provide a way to get us back on track. Aside from their contributions to my research, their ongoing kindness and guidance is something I am highly appreciative of.

I would also like to thank the other member of my supervisory committee, Chris Beck. Chris' careful reading and revision of my work greatly improved this thesis, and beyond that his challenging questions were crucial to me completing a thorough doctoral education. His interesting insights and ideas always inspired me to think about what improvements I could make and what follow-up research would be most useful.

I would also like to thank Peter van Beek for agreeing to act as my external advisor, and for his excellent suggestions and revisions which further improved this thesis.

I am grateful to all of the people I collaborated with, both formally and informally, including other graduate students, other members of the SAT community, and other professors at the University of Toronto. There are too many names to list individually, but each of them had an impact on my development and my ability to write this thesis.

Last but certainly not least, I would like to thank my family and friends, especially my mom, my dad, my sister, and my wife, for providing me with everything I need in life to succeed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Boolean satisfiability (SAT) is a fundamental problem in computer science which seeks to determine whether or not any truth assignment satisfies a propositional logic formula. Many problems in computer science can be solved efficiently by converting them into propositional logic formulas and then solving them with a modern SAT solver [1]. Over the last two decades, state-of-the-art SAT solvers have gone from solving real-world problems with hundreds of variables to solving real-world problems with millions of variables. Because of this progress, there are increasingly many applications of SAT solvers, including software and hardware verification [2], verifying robustness of machine learning models [3], proving theorems from combinatorial mathematics [4], and many others. Apart from real-world applications, SAT solvers are an important component of many other tools used to solve computationally hard problems, including Bounded Model Checkers (BMC) [5], Model Counters (#SAT) [6], SAT Modulo Theory (SMT) solvers [7], pseudo-boolean solvers [8], and more. This makes improving the efficiency of SAT solvers incredibly important, since it helps each of these tools become more efficient as well. In the words of a pioneer in computing, Donald Knuth [9]:

> "The SAT problem is evidently a killer app, because it is key to the solution of so many other problems."

Improving SAT solvers is difficult, which has been made apparent throughout the last two decades of the SAT Competitions where often the best solver wins by solving at most only a few more instances than its competitors. This is perhaps summed up best by the following quote from Gilles Audemard and Laurent Simon, two pioneers in the field [10].

> "We must also say, as a preliminary, that improving SAT solvers is often a cruel world. To give an idea, improving a solver by solving at least ten more instances (on a fixed set of benchmarks of a competition) is generally showing a critical new feature. In general, the winner of a competition is decided based on a couple of additional solved benchmarks."

Yet, these small incremental improvements made to SAT solvers year after year, along with some true breakthroughs every once in a while, gradually accumulate and lead to massive improvements in the capability of solvers over a longer time frame. This does not only apply to SAT solving, but extends to other areas of propositional reasoning as well. Take, for example, MaxSAT solving [11, 12], which plays an important role

in this thesis, where the jumps in the number of problems solved have been even larger than those realized in SAT solving[1]. Although it is difficult to advance the state-of-the-art MaxSAT solvers with one new idea, since the MaxSAT Competitions are usually decided by only a handful of solved instances each year, the cumulative improvements have added up over the last decade or so to make massive differences. None of the MaxSAT solvers that were competitive from 10-15 years ago would be competitive with the state-of-the-art solvers today on any broadly representative set of benchmarks.

Redundant Work

Much of the work done in SAT solvers is repetitive, especially when working on formulas generated by real-world problems which tend to have symmetries, patterns, and structure. Lots of influential work has been done over the last few decades to address these inefficiencies and remove some of the redundancy, including clause learning [13], various forms of preprocessing [14], incremental SAT solving techniques [15], and others. However, in spite of this progress, there still remains a lot of repetitive computation done even in state-of-the-art modern SAT solvers. Additionally, SAT solvers perform unit propagations and other operations extremely quickly, but much of this information is discarded and only a tiny amount is used to compute heuristics that direct the solver in the future. *The main issue with this repetitive work is that it can show up very often throughout the execution of the solver, resulting in a dramatic slowdown, and making problems which could otherwise be tackled by the solver become infeasible in a realistic amount of time and memory.*

This redundant work we are referring to can be best illustrated by tracing through the execution of a propositional logic solver and observing what computations are being made and what computational results are being discarded. Some examples of redundant work that can be observed when tracing through the execution of propositional logic solvers include extracting the same unsatisfiable core multiple times (in the context of MaxSAT), reconstructing the same sequence of inferred variable assignments over and over again (in the context of SAT), and searching through the same clauses multiple times after each assumption (in the context of incremental SAT). These are among the issues that we set out to address in this thesis.

1.1.1 Objective

There are two main ways we will reduce the redundant work performed by SAT solvers. The first way involves exploiting previous computations the solver made in order to prevent the solver from having to re-perform those computations over and over again. The second way is to notice some redundant work that solvers generally perform and prevent it ahead of time, such as by altering the order that computations are done in. The objective of this thesis is to introduce new techniques to improve the performance of SAT solvers, and other related propositional logic solvers, by both exploiting previous computations the solver has made and eliminating redundant work the solver will perform in the future.

We will not limit ourselves to only pure SAT solving in this thesis, because the ability to reuse computations and avoid redundant work also extends to other propositional logic solvers, such as MaxSAT solvers [12], MUS extractors [16], incremental SAT, and incremental MaxSAT [17]. Although each of those propositional logic solvers is built on top of a SAT solver and can benefit from the SAT techniques we introduce, some other techniques we introduce will not necessarily operate in the SAT solver directly, but will reduce

[1]The slides presented at the 2018 MaxSAT Evaluation (https://maxsat-evaluations.github.io/2018/mse18-talk.pdf) have more discussion on this.

some of the redundancy that occurs outside of the SAT solver as well.

Thesis Statement: Techniques to reduce redundant work in a propositional logic solver can dramatically improve the performance of the solver.

1.2 Scope

Although we are mainly dealing with improving SAT solvers and other propositional reasoning solvers related to SAT, this thesis can be of interest to other communities, particularly in other areas of automated reasoning or operations research. Of course, because there are so many applications of SAT solving, the improvements we make in this thesis could lead to improvements in other areas that rely on the performance of these solvers. However, we are mainly trying to improve the performance of these propositional reasoning solvers themselves on a wide range of benchmarks, rather than focusing on any specific type of benchmarks or applications.

The work in this thesis is very empirically based, meaning we evaluate the success or failure of our ideas by their performance, i.e., how well they improve the number of problems we can solve with state-of-the-art solvers. It may seem as if there is too much emphasis on improving results in competition settings on benchmarks used in the competitions, whether it is the SAT competition or the MaxSAT competition, throughout each chapter of this thesis. While the competitions themselves are not inherently important, they provide a setting that is fair for all solvers to compare against each other. Furthermore, the competition benchmarks come from a wide range of applications which have been picked by a group of experts specific to the field and deemed important and relevant. Our purpose is to introduce techniques which will be useful beyond the competition setting, hopefully in real-world and industrial settings to solve new problems that would otherwise not be solved, but the competition settings and benchmarks provide us the fairest way to test and compare the effectiveness of our ideas against alternatives.

We always prove the soundness of each technique we introduce, as it is important that the techniques not only work in practice but also in theory. This is because the rapid advances in the capabilities of propositional reasoning solvers means that larger and more difficult instances will be of interest in the near future. By proving the soundness of our techniques now, we reduce the chance that our techniques cause bugs in the future. Although we will introduce some interesting propositions that require proving at times, and we will have to prove other non-trivial properties about the techniques we introduce, we do not expect this work to be as relevant to advancing the theoretical computer science literature. However, just as theory inspires empirical work, empirical work can inspire theory, and so some of the ideas in this thesis may also be of interest for theoreticians to the effect that they might inspire them to fill gaps we left open, or come up with new theorems that relate to our work.

1.3 Summary of Contributions

Below is a brief summary of the research contributions we have made while developing this thesis. We will first list the published papers that are relevant before going on to describe the contributions that will appear in each chapter of this thesis.

1.3.1 Relevant Publications

- Randy Hickey and Fahiem Bacchus. "Speeding up assumption-based SAT." Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019.
- Randy Hickey and Fahiem Bacchus. "Trail saving on backtrack." Theory and Applications of Satisfiability Testing–SAT 2020: 23rd International Conference, Alghero, Italy, July 3–10, 2020..
- Randy Hickey and Fahiem Bacchus. "Large Neighbourhood Search for Anytime MaxSAT Solving." 31st International Joint Conference on Artificial Intelligence-IJCAI 2022, Vienna, Austria, July 23-29, 2022.

1.3.2 Speeding Up Assumption-Based SAT (Chapter 3)

Assumption-based SAT solving is an essential tool in many applications of SAT solving, especially in incremental SAT solving. For example, assumption-based SAT solving is used when solving MaxSAT, when computing minimal unsatisfiable subsets and minimal correction sets, and in various inductive verification applications. The minisat SAT solver [18] introduced a simple technique for extending a SAT solver to allow it to handle assumptions by forcing the SAT solver to make the assumed literals its initial decisions. This approach persisted in almost all SAT solvers making it the most commonly used technique for handling assumptions. We will explain some deficiencies in this approach that can hinder its efficiency, and provide a very simple modification that fixes these deficiencies. We will also examine the issue of repeated work when the solver backtracks over the assumptions, e.g., on restarts or when a new unit clause is learnt, and develop a new method for avoiding this repeated work that addresses some deficiencies of prior approaches. This technique managed to improve the state-of-the-art solvers from two different applications of assumption-based SAT: MUS extraction and MaxSAT solving.

1.3.3 Trail Saving on Backtrack (Chapter 4)

Part of the work in speeding up assumptions dealt with saving portions of the trail in the assumptions decision levels in order to re-use them later. However, there was still no way to re-use portions of the trail from outside of the assumption decision levels. This inspired us to extend the idea of "trail savings" to save portions of the trail more generally at all decision levels for regular SAT solving (i.e., non-incremental SAT solving).

A CDCL SAT solver can backtrack a large distance when it learns a new clause, e.g, when the new learnt clause is a unit clause, the solver has to backtrack to level zero. When the length of the backtrack is large, the solver can end up reproducing many of the same decisions and propagations when it redescends the search tree. Different techniques have been proposed to reduce this potential redundancy, e.g., partial/chronological backtracking and trail reusage on restarts. In this chapter we present a new trail saving technique that is not restricted to restarts, unlike prior trail saving methods. Our technique makes a copy of the part of the trail that is backtracked over. This saved copy can then be used to improve the efficiency of the solver's subsequent redescent. Furthermore, the saved trail also provides the solver with the ability to look ahead along the previous trail which can be exploited to improve its efficiency. Our new trail saving technique offers different tradeoffs in comparison with chronological backtracking and often yields superior performance. We also show that our technique is able to improve the performance of several state-of-the-art solvers.

1.3.4 Large Neighbourhood Search for Anytime MaxSAT Solving (Chapter 5)

Moving beyond SAT solving and incremental SAT solving, another area that has algorithms with many repetitive computations is anytime MaxSAT solving, where the aim is to return a good (but not necessarily optimal) solution in a short amount of time. Many of the algorithms used for anytime MaxSAT solving operate by continually taking the best solution found so far and trying to improve it slightly. This often results in finding multiple solutions to the MaxSAT problem that are very similar to one another (i.e., the truth assignments are the same for most of the variables in each solution). By formulating an algorithm as an instance of incremental MaxSAT (where a MaxSAT solver is called iteratively on a group of very similar MaxSAT problems), one could more easily try to exploit some of this repetition. This led us to coming up with a new algorithm for anytime MaxSAT solving, using a well-known technique called Large Neighbourhood Search (LNS) [19, 20], and later we will see how this algorithm can be viewed as an instance of incremental MaxSAT.

We will present this method [21] for applying LNS to improve anytime (MaxSAT) solving by introducing a neighbourhood selection policy which shows good empirical performance. We will show that our LNS solver can often improve the suboptimal solutions produced by other anytime MaxSAT solvers. When starting with a suboptimal solution of reasonable quality, our approach often finds a better solution than the original anytime solver can achieve. We will demonstrate that implementing our LNS solver on top of three different state-of-the-art anytime solvers improves the anytime performance of all three solvers within the standard time limit used in the incomplete tracks of the annual MaxSAT Evaluation. Finally, we will show how LNS for MaxSAT can be viewed as an instance of incremental MaxSAT and exploit some of the repetition between MaxSAT calls.

1.4 Impact of Contributions

Some of the ideas in this thesis have been applied successfully by others since their original publication. The assumptions techniques described in Chapter 3 were implemented in UWrMaxSat [22], the winner of the 2020 weighted track of the MaxSAT evaluation, while trail saving from Chapter 4 has been extended to SMT solving [23] and also seems to have helped inspire ideas used in Incremental Lazy Backtracking [24]. However, many of the best solvers have not yet implemented these ideas and there remains great potential to extend some of them to other applications. Our own experiments confirm that using LNS, as detailed in Chapter 5, still improves the state-of-the-art anytime MaxSAT solvers, but follow-up work attempting other possible neighbourhoods is needed.

1.5 Dissertation Outline

The first chapter of this thesis, the introduction, provided motivation for why the problem of redundant work in propositional logic solvers is worth addressing, summarized the contributions we have made, and gave an outline of the remainder of the dissertation.

The second chapter of this thesis will give some general background that will be useful when reading the remainder of this thesis. Although a more detailed background section containing definitions and notation will appear within each of the technical chapters, the second chapter will introduce some key concepts of modern SAT solving and MaxSAT solving algorithms for the benefit of the reader.

The next three chapters of this thesis will be technical chapters which will outline the main contributions

that the published research from this thesis has produced. The third chapter will be devoted to a method we proposed [25] to speed up the unit propagation of assumptions by avoiding repetitive work in the unit propagation process. The fourth chapter will be devoted to trail saving [26], which automatically restores some of the unit propagation steps performed before the last backtrack. The fifth chapter of this thesis will be devoted to applying a Large Neighbourhood Search (LNS) algorithm [20] to "anytime" MaxSAT [21]. In each of these technical chapters, we will describe the new technique we are proposing in detail, prove its soundness, provide pseudocode for its efficient implementation, and show empirically that it improves multiple state-of-the-art solvers from the domain it is being applied to. We also made the code implementing each of our ideas freely available to download.

Finally, in the last chapter of this thesis we will make some concluding remarks about our findings and provide some potential ideas for future research.

Chapter 2

Background

2.1 SAT Solving

Satisfiability is the problem of determining whether or not there is an assignment to each of the variables in a logic formula which makes it evaluate to *true*. However, a lot of the attention historically in satisfiability has been restricted to the satisfiability of propositional logic formulas, and for the remainder of this thesis whenever we refer to "satisfiability" or "SAT ", we are referring to the satisfiability of a propositional logic problem. Furthermore, most of the focus within propositional satisfiability has been on the satisfiability of logic formulas in Conjunctive Normal Form (CNF). Whenever we refer to a SAT problem, a SAT instance, a CNF, or a SAT formula, we are referring to a Boolean logic formula in conjunctive normal form.

We will now define what a CNF formula is. As we are dealing with Boolean logic formulas, all variables are Boolean. A **literal** is a Boolean variable (which is called a positive literal) or its negation (which is called a negative literal). A **clause** is the fundamental building block of CNF formulas, and it is defined to be a disjunction of literals (and only a disjunction of literals). A clause can contain no literals, in which case it is considered the **empty clause**, and always evaluates to false. Finally, a **CNF** formula is defined to be a conjunction of clauses (and only a conjunction of clauses, meaning no other logical connectives are permitted). A CNF formula can contain no clauses, in which case it is considered vacuously true.

It is sometimes useful to refer to a formula as a set of clauses, and it is to be understood that this set represents a conjunction of the clauses it contains. Similarly, it is sometimes useful to refer to a clause as a set of literals, and it is to be understood that this set represents a disjunction of the literals it contains. Going a step further, it can sometimes also be useful to refer to a clause as an ordered list of literals, and it is to be understood that this ordered list not only represents the underlying clause (i.e., a disjunction of its literals), but also puts an ordering on the literals. This is sometimes useful for referring to the order in which certain algorithms will access each literal. For this we will use parentheses around a list of literals; i.e. (x, y) represents the clause $x \vee y$ and more specifically represents an ordering of the literals where x is before y.

An **assignment** to a SAT problem gives a truth value to every variable (i.e., each variable can only be true or false). It is sometimes convenient to discuss the truth value of a literal as well, and it should be understood that the truth value of a positive literal is equal to the truth value assigned to its underlying variable and the truth value of a negative literal is equal to the negation of the truth value of its underlying variable. A **solution** to a SAT problem is a truth assignment which makes the formula evaluate to true (i.e., makes every clause in the formula evaluate to true). There can be multiple possible solutions to a particular formula.

SAT is NP-complete [27], meaning all NP-complete problems can be encoded into SAT in polynomial time. While there is no known polynomial time algorithm for SAT, as mentioned above, SAT solvers are very fast on most industrial instances, thanks in large part to the conflict-driven clause learning algorithm, which we will discuss next. Aside from industrial instances, random instances and specially crafted instances containing only a few hundred variables can be intractable for any known SAT solver. Aside from NP-complete problems, SAT solvers are sometimes used to solve problems beyond NP, especially when used in conjunction with another algorithm or when called repeatedly as in incremental SAT applications, which we will discuss later.

2.1.1 CDCL algorithm

The main algorithm used in state-of-the-art SAT solvers, and used by the majority of applications which use SAT solvers, is the conflict-driven clause learning (CDCL) algorithm. Unless otherwise stated, it should be assumed for the remainder of this thesis that whenever we refer to a SAT solver, we are referring to a SAT solver that uses the CDCL algorithm. We will review some of the concepts of the CDCL algorithm below, but a more complete source to read about CDCL is the chapter on that topic [28] in the Handbook of Satisfiability. Some other SAT algorithms are relevant to the field and work on very specific types of instances, but are not as relevant to this thesis. These algorithms include stochastic local search, lookahead solvers, binary-decision-diagram based algorithms, and others. A good source to read about these algorithms is the Handbook of Satisfiability [7].

CDCL solvers take CNF formulas as input and start with a **preprocessing** phase, where techniques to simplify the formula are executed before the main search begins. These techniques include bounded variable elimination, subsumption, and others. The preprocessing phase is also where the simplification of the CNF formula happens, such as eliminating tautologies and making variable assignments which are trivially forced. For a more thorough overview of preprocessing in SAT solvers, consult the Handbook of Satisfiability [14].

Next is the main search, where the algorithm will assign truth values to variables either 1) by performing unit propagation or 2) by making decisions. A **decision** is when a variable is selected by the solver's heuristics and assigned according to the solver's policy, as opposed to being forced (which we will define shortly), and typically solvers will have to make many decisions before satisfiability of the formula can be determined. A partial truth assignment is maintained via a **trail**, which is an ordered list of all literals that are made true in the order they were assigned (by either decision or by unit propagation). The trail is a critical data structure, and might be accessed billions of times before satisfiability of the formula is determined, thus maintaining its efficiency is of utmost importance throughout the introduction of all techniques in this thesis.

Whenever a clause c has all of its literals except one (let that literal be l) falsified by the current partial assignment, it is called **unit** (or a *unit clause*), and the solver will assign l to true. We refer to this process as l being **forced** (or **unit propagated**), and we call c a **reason** for l when c is the clause that became unit to force l. Reasons for each propagated literal are important to keep track of because they help us learn clauses during conflict analysis, as we will discuss shortly.

The entire process of detecting unit clauses and forcing the appropriate literals is referred to as **unit propagation** and it is one of the most critical parts of the solver to keep running as fast as possible. SAT solvers can perform millions of propagations per second, and we will try to maintain the speed or even improve the speed of unit propagation when introducing any new techniques throughout this thesis. Instead of searching through all clauses to detect which clauses are unit during unit propagation, what modern solvers do is mark two unassigned literals in every clause to be **watchers** of the clause, and then only check if a clause

is unit when one of these watchers becomes false. This is known as the two-watched-literal scheme and was introduced in the Chaff solver [29].

When there are no more unit clauses in the formula, unit propagation is over and the algorithm then makes a new decision (i.e. assigns a variable by decision). Every assigned variable is given a **decision level**, which is defined as equal to the number of decisions on the trail up to and including itself. Note this means that all propagated literals also have a decision level, and whenever a decision d forces a literal l, d and l will have the same decision level.

Next, we will talk about what happens when the solver makes decisions that can not possibly lead to a solution. Whenever a clause has all of its literals falsified by the current partial assignment, it is referred to as the **conflict**, and we say that the solver (or the trail) is under conflict (alternatively, we can say in contradiction or has reached a dead-end). This is where clause learning, the main feature of CDCL, happens. The conflict is then resolved[1] against the reasons of literals appearing before it on the trail up to some fixed point. The final resolvent is called a **learnt clause** and is added to the formula. This learnt clause will block all future partial assignments which falsify it from being explored, saving potentially exponentially many explorations. Learnt clauses [13] form the basis of the CDCL algorithm, and their introduction is arguably the most important component [30] of modern solvers that enable their efficiency.

In order to undo the conflict (i.e., restore the current partial assignment to a point where the trail was not under conflict), a **backtrack** must occur, where some number of literals are unassigned. Once a literal is unassigned, it no longer has a reason clause attached to it. When backtracking, at least all of the literals on the trail from the current decision level are unassigned (and often times more than just the current decision level will be unassigned, depending on the learnt clause and the type of learning scheme the solver is using). During a backtrack, either all of the literals on a decision level must be unassigned or none, and before all literals from a decision level are unassigned, all literals from greater decision levels must be unassigned first.

A **restart** is triggered periodically by the solver and forces the solver to backtrack all the way to decision level 0 after a conflict. The solver's heuristics will determine when a restart is done, with the purpose being to give the solver a chance to avoid searching in one particular part of the search space for too long. Frequent restarts have been shown to be beneficial in practice. For a detailed discussion on restarts and their effectiveness, consult the Handbook of Satisfiability [28].

Eventually as the solver continues to execute its main search and find conflicts, too many learnt clauses are saved which can slow down unit propagation or use too much memory. The reason too many learnt clauses can slow down unit propagation is because when unit propagating a literal l, all clauses on l's watchlist need to be accessed. Even in practice, exponentially many learnt clauses can be collected before the satisfiability of a formula is determined. Thus periodically, some of the learnt clauses need to be deleted and this is referred to as **garbage collection** [31]. It should be noted that garbage collection technically makes the CDCL algorithm incomplete[2], but in practice it is necessary. The most common metric for deciding which clauses to keep and which to delete is based on **Literal Block Distance (LBD)** score, which was first proposed in the paper that introduced Glucose [32]. The decision level of each literal in a learnt clause is counted, and the number of different decision levels for that learnt clause is its LBD score. The clauses with higher LBD scores are deleted before the clauses with lower LBD scores.

[1] Two clauses are resolved by the resolution rule in logic. The resolution of clause $c_1 = (x_1 \vee A)$ with clause $c_2 = (\neg x_1 \vee B)$, where A and B can be any disjunction of literals, results in a resolvent $A \vee B$.

[2] The algorithm becomes incomplete because there may exist a group of learnt clauses which are all needed simultaneously to prove unsatisfiability, but this group might never have a chance to exist simultaneously if one or more of these clauses continue to be deleted during garbage collection.

2.2 Assumption-Based SAT

Een and Sorensson [18] introduced a SAT solver called minisat in 2003, which became the most widely used SAT solver of its time and is still used in some applications today. Among other things, minisat implemented the Variable State Independent Decaying Sum (VSIDS) variable selection heuristic in a much more efficient way, was arguably the first CDCL solver to have a clean interface that allowed for simple extensions, and introduced the idea of assumptions to act as an incremental SAT interface. The latter contribution, assumptions, is what we will describe below. Assumptions play an important role in multiple parts of this thesis.

In incremental SAT solving, a sequence of very similar SAT formulas are solved one after another in what we will refer to as **episodes** or **invocations**. Since the formulas are very similar, many of the learnt clauses from one formula can be useful for solving subsequent formulas.

If the sequence of formulas are such that new clauses are added in between episodes but no clauses are ever removed (so that each formula builds on the last one), learnt clauses may be reused; i.e., if a learnt clause is implied by a formula F, it is also implied by $F \wedge C$, where C is a conjunction of new clauses. However, many applications do not only add clauses but also require temporarily fixing the truth values of a set of variables in one episode and then fixing the truth values of a different set of variables in the next episode. For example, suppose in one episode the user wishes to solve $F \wedge A_1$, where A_1 is a conjunction of some literals, and then wishes to solve $F \wedge A_2$ in the next episode. If A_1 is not a subset of A_2 and the user simply adds the literals in A_1 as unit clauses to F, then clauses learned from $F \wedge A_1$ would not necessarily be implied by $F \wedge A_2$, so they would have to be discarded.

Assumptions solve this problem. The user supplies the list of literals that they want to be forced to be true for this episode and the solver then uses these assumption literals as its initial decisions. Thus, the SAT solver ensures that these literals are made true as required by the user. After making all of the assumptions true, the solver is then free to decide on the other variables. If while making the assumptions true the solver runs into a conflict, it will return that the episode is "unsatisfiable under assumptions". It can then do conflict analysis (i.e. a series of resolutions) to compute a **final conflict clause** consisting of only negated assumption literals. The final conflict clause is entailed by the formula, so it contains the negations of those assumptions which were unable to be jointly satisfied along with the formula. Since the assumptions were never added to the root formula as unit clauses, all learnt clauses remain valid for the next episode [3].

Assumptions are also used in conjunction with **relaxation variables** to activate or deactivate a clause during incremental SAT. For a clause c, a **relaxation variable** r is one such that it can be added to the clause to form a new clause $c_r = c \vee r$ that replaces c during a particular invocation. In order to activate this clause during an invocation, r would be assumed false (i.e., we usually say we assume $\neg r$), thus forcing the original clause c to be made true during this invocation (in order for this invocation to be satisfiable). In order to deactivate this clause during an invocation, r would be assumed true, thus c would not necessarily be forced to be true in order for this invocation to be satisfiable.

It is clear that the idea of using assumptions was incredibly successful since the majority of incremental SAT applications either used minisat directly or implemented minisat-like assumptions into a different SAT solver. There have been attempts since then to improve the way in which assumptions are handled by SAT solvers, which we will discuss in future chapters, and one of the fundamental contributions of this thesis is to improve the efficiency of unit propagating assumptions (discussed in Chapter 3).

[3] When the assumptions are decisions, each learnt clause will be entailed by F (and possibly contain assumptions). If, instead, the assumptions had been forced and added to the formula, then each learnt clause would be entailed by $F \wedge A$ but not necessarily entailed by F, and thus may not be eligible to be reused in future episodes.

2.3 MaxSAT Solving

MaxSAT plays an important part in multiple chapters of this thesis. Classical MaxSAT is the problem of determining the maximum number of clauses which can be satisfied by any possible truth assignment to a CNF formula. It is often the case that we want not only the number of clauses that can be satisfied, but also the particular truth assignment that satisfies that number of clauses. MaxSAT solvers allow us to solve optimization problems while taking advantage of SAT solving technology, since SAT solvers are used frequently inside most MaxSAT solvers.

However, modern MaxSAT solving is mainly focused on what was traditionally known as partial MaxSAT[4]. In partial maxsat, there are two sets of clauses, the **hard clauses**, which must be satisfied by the solution, and the **soft clauses**, which do not necessarily have to be satisfied by the solution. The problem then becomes to find the assignment that satisfies the maximum number of soft clauses while satisfying all of the hard clauses. All instances used in this thesis will have both hard clauses and soft clauses, unless otherwise specified.

Modern MaxSAT solving is also becoming increasingly focused on what was traditionally known as weighted partial MaxSAT[5]. In weighted partial MaxSAT, each soft clause is given a real-numbered weight and the problem is to determine the truth assignment that maximizes the total satisfied weight (where satisfied weight refers to the sum of the weights of the clauses that are satisfied), as opposed to just satisfying the maximum number of clauses. It should be understood that whenever we refer to MaxSAT in this thesis, we are referring to partial weighted MaxSAT, unless otherwise stated. Unweighted instances still play an important role in analyzing data, and thus we will separate weighted and unweighted instances where appropriate. A truth assignment that satisfies all of the hard clauses is called a **feasible solution**.

Partial MaxSAT is FP^{NP}-complete, meaning MaxSAT can be solved with a polynomial number of calls to an NP-oracle (e.g., a SAT solver) and any problem in the FP^{NP} class can be encoded into MaxSAT. In fact, unweighted MaxSAT can actually be solved with a logarithmic number of calls to an NP oracle [33], however this is not the case when we introduce weights.

An equivalent way of defining MaxSAT is to determine the truth assignment that minimizes the falsified weight (where falsified weight refers to the sum of the weights of the clauses that are falsified) instead of maximizing the satisfied weight. Because of the convention in the field to define MaxSAT this way, this is the definition we will use as well. It is more convenient under this definition to replace the word weight with **cost**, which is what we will usually do as well. Similarly, when we refer to an **optimal solution** in the context of MaxSAT, it should be understood that we are referring to both the particular assignment that falsifies the minimum total cost along with that total cost itself. It should be noted that there can be multiple optimal solutions to a MaxSAT problem.

A MaxSAT solver is called a **complete solver** when it is guaranteed to terminate with an optimal solution, and it is called an **incomplete solver** otherwise. Incomplete solvers are still useful in some contexts, as they aim to more rapidly find low cost solutions that are not necessarily optimal. For some instances, it is almost impossible to find an optimal solution in a reasonable amount of time, so finding a good solution can be better than not finding one at all. **Anytime** MaxSAT solving aims to find the lowest cost solution possible within a given time limit. Incomplete solvers are typically more useful in the context of anytime solving.

Many MaxSAT algorithms make use of a SAT solver, usually by **relaxing** the MaxSAT instance in some way to become a SAT instance, and solving it with a SAT solver. For most of those algorithms, many such SAT instances would be solved, so the use of assumptions makes things more efficient. Relaxing the MaxSAT

[4]This is not true in theoretical computer science, as the majority of MaxSAT literature there is still focused on classical MaxSAT.
[5]As previously noted, this is not the case in theoretical computer science.

formula into a SAT formula commonly involves removing some of the soft clauses in the MaxSAT formula according to some policy and then making the remainder of the soft clauses hard. For example, a MaxSAT formula with hard clauses H and soft clauses $S = \{s_1, s_2, s_3, ..., s_n\}$ that gets **relaxed by removing** s_1 and s_2 would become the SAT formula $H \cup \{s_3, ..., s_n\}$, where the costs of all soft clauses are ignored in the new SAT formula.

Other algorithms relax the MaxSAT formula into a SAT formula not by ignoring some of the soft clauses as above, but instead by restricting the relaxed SAT formula with the addition of cardinality constraints. A **cardinality constraint** [8] in our context is a logical formula which restricts the number of variables that can be assigned a particular truth value to some fixed number, and this logical formula is then converted into clauses. For example, a cardinality constraint which forces at most 1 of the variables x, y, z to be assigned true could be represented by the clauses $\{x, \neg y, \neg z\}, \{y, \neg x, \neg z\}, \{z, \neg x, \neg y\}$. We will go over some of the main algorithms used in the MaxSAT solvers that are relevant to this thesis below.

2.3.1 MaxSAT algorithms

The two main complete algorithms for MaxSAT that will be discussed will be **implicit hitting-set based** algorithms and **core-guided** algorithms. For a complete overview of these algorithms, a good source to consult is the Maximum Satisfiability chapter in the Handbook of Satisfiability [12]. Branch-and-bound is a third complete algorithm for MaxSAT, but will not be as relevant to this dissertation.

Both the implicit-hitting set algorithm and the core-guided algorithms rely on the concept of an **unsatisfiable core**. An unsatisfiable core (also sometimes referred to as just a "core" for simplicity) of a SAT formula is a subformula (i.e., a set of clauses where each clause is contained in the SAT formula) that is itself unsatisfiable. By definition, this means a satisfiable formula has no cores. In the context of MaxSAT, when an unsatisfiable core is referred to, it is often understood that we are only referring to the set of soft clauses that are contained within the core.[6] For the remainder of this thesis, when we refer to a **core** or **unsatisfiable core**, we are only referring to the set of soft clauses contained in the unsatisfiable core. An unsatisfiable formula can have many different cores. A core can be extracted from a final conflict clause, since algorithms which are using cores would only make assumptions on relaxation variables (i.e., variables used to activate or de-activate a clause, as discussed in Assumption-Based SAT above). Any superset of a core is also a core. Thus, we are usually interested in obtaining minimal cores when possible, and a core which is a superset of another core is not desirable (since it does not provide any new information).

The implicit-hitting set algorithm relies on repeatedly relaxing the MaxSAT formula into a SAT formula by removing a minimum cost hitting set of the collection of found cores, until the relaxed SAT formula is satisfiable. Each time the relaxed SAT formula returns unsatisfiable, a core is extracted and added to the collection of cores. A **hitting set of cores** is a set of soft clauses which covers each core (by covering, it is meant that at least one clause in the core also occurs in the hitting set). A **minimum cost hitting-set of a set of cores** is the set of soft clauses that has the lowest combined cost and covers every core in the set of cores. Once the relaxed SAT formula is satisfiable, the solution to the relaxed SAT formula corresponds to a solution to the MaxSAT formula (i.e., the corresponding MaxSAT solution has the same truth assignment for each variable that the relaxed SAT solution has). Optimizations and different variations exist for the two main solvers that have used this approach, MaxHS [34] and LMHS [35].

[6] The reason we exclude the hard clauses from an unsatisfiable core is for convenience, because the algorithms used to "relax" cores in MaxSAT do not operate on any hard clauses. Of course, the set of soft clauses of a core alone are not necessarily unsatisfiable, but the union of the set of soft clauses of a core with the set of all hard clauses is guaranteed to be unsatisfiable.

In the core-guided algorithms, a SAT solver is repeatedly called on a relaxed SAT formula that continues to change each iteration. When the relaxed SAT formula is unsatisfiable and returns a core, a cardinality constraint allowing one of the clauses in that core to become false is added to the formula, and the process is repeated until enough of the formula has been restricted for the SAT solver to solve the relaxed SAT formula. Similarly to the implicit-hitting-set algorithm, the solution to the relaxed SAT formula corresponds to a solution to the MaxSAT formula. There are several variations of core-guided algorithms that have been successful, including MSU3 [36], OLL [37], and others.

The two main incomplete algorithms we will discuss in this thesis are stochastic local search and linear-sat-unsat search (LSU). There are many other heuristics and algorithms, but most of the competitive anytime solvers use the linear-sat-unsat algorithm as a base for their solver (or call another solver at some point that uses linear-sat-unsat search). Several of the solvers also use stochastic local search at some point in their execution.

In the linear-sat-unsat search (LSU) algorithm [38], a SAT solver is called with all of the soft clauses in the formula relaxed at first, to get a feasible solution. After that, a cardinality constraint is added to the formula so that another solution can be found whose cost is better (i.e., lower) than the last solution found. This continues until the SAT solver returns unsatisfiable, at which time the solution can not be improved any further. There are also many heuristics and improvements on top of LSU that have been developed over the years, including boolean multilevel optimization (BMO) [39], polarity selection heuristics [40], and others. Although the algorithm described above is technically complete, some of these improvements do more than just searching for a better solution each iteration, and as a result make the algorithm incomplete.

In stochastic local search [41], a clause is typically chosen and, subsequently, a variable within that clause is chosen. This variable's value is then **flipped**, which means if it was assigned true it becomes false, and if it was assigned false it becomes true. The selections of the clause and variable to flip depend on heuristics, but usually one is looking to satisfy more clauses with the variable flip (or in the MaxSAT case, trying to satisfy a group of clauses with greater total weight). The main implementation of stochastic local search that is used by the best anytime MaxSAT solvers is satlike [42]. Satlike makes several modifications to the stochastic local search algorithm to make it more appropriate for MaxSAT, including focusing mainly on soft clauses instead of hard clauses at first, and then gradually increasing the consideration of the hard clauses when feasible solutions are not being found.

2.4 Cactus Plots

For many years in the literature, cactus plots were typically used in publications to compare SAT solvers to one another (and also, to compare MaxSAT solvers to one another). Recently, there has been a consensus to switch to cumulative distribution function plots (which flips the "Time" and "Instances Solved" axes and aligns more with the standard in other fields). However, at the time of publication of most of the work in this thesis, cactus plots were the standard, so that is what we will use to display our results in the technical chapters.

In a cactus plot, the total number of instances solved is along the x-axis, and the time taken to solve those number of instances is along the y-axis. First, all problems are sorted by the amount of time they took to solve (from smallest to largest). Then for the nth problem solved from a set of benchmarks, another point on the graph is made with the y coordinate equalling the time it took to solve all n problems (where each problem is solved on its own). This can sometimes cause confusion from those outside of the field looking

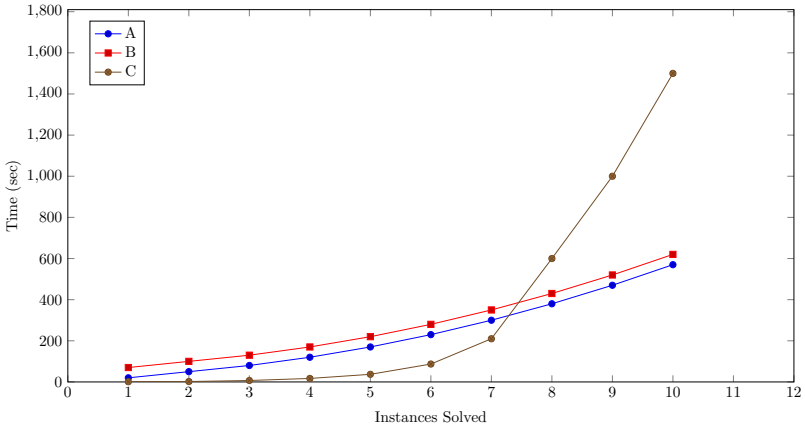

Figure 2.1: Example cactus plot for solvers named A, B, and C. C solves its first 7 problems faster than A and B solve their first 7 problems, respectively, but C takes longer than A and B to solve more than 7 problems. A solves n problems faster than B for all n.

at graphs wondering which solver performs best. The solver whose graph is more quickly moving along the x-axis than it is along the y-axis (i.e., the graph with a slope closer to 0) is performing better. In the example cactus plot in Figure 2.1, A is performing better than B the entire way; C starts off performing better than both A and B for solving ≤ 7 instances, but ends up performing worse than both A and B for solving ≥ 8 instances.

Chapter 3

Speeding Up Assumption-Based SAT

3.1 Introduction

3.1.1 Motivation

A wide range of applications of SAT solving rely on assumption-based incremental SAT solving. This includes algorithms for bounded model checking (BMC), e.g., [43, 44, 45, 46]; minimum unsatisfiable set (MUSes) extraction, e.g., [47, 48, 16, 49], computing minimal correction sets (MCSes), e.g., [50, 47, 51, 52]; and solving maximum satisfiability (MaxSAT), e.g., [53, 54, 55, 56, 57, 58, 59].

Assumption-based SAT involves requesting the SAT solver to find a solution that also satisfies a specified set of assumptions, encoded as a conjunction of literals. Assumptions are particularly useful in *incremental* SAT solving, where the SAT solver is called on a sequence of formulas that are closely related to each other. Each formula could be solved by invoking a new instance of the SAT solver; but then information computed during one solving invocation (e.g., learnt clauses) cannot easily be exploited in subsequent solving invocations. The idea of incremental SAT solving is to use only one instance of the SAT solver for all of the formulas so that all information computed when solving the previous formulas can be retained to make solving the next formula more efficient. In this case, we can monotonically add clauses to the SAT solver or use different sets of assumptions to specify each new formula to be solved. With assumptions, for example, we can add or remove certain clauses by adding to those clauses a new literal ℓ. When we assume $\neg\ell$ these clauses become active (added to the formula), and when we assume ℓ these clauses become inactive (removed from the formula).

The most common technique for supporting assumptions in SAT solvers was proposed in a paper by Eén and Sörensson [46] and implemented in the minisat solver [60]. This technique involves forcing the SAT solver to make as its initial decisions the assumed literals. This approach is still used in the most commonly available SAT solvers handling assumptions, including minisat [60], Glucose [61], Lingeling [62], and CryptoMiniSat [63]. However, as we will explain in detail later, this approach can suffer from unnecessary overhead when enforcing the assumptions. In particular, a clause which is made unit by the assumptions may have to be accessed numerous times (on the order of the number of literals that the clause contains) before the unit propagation process recognizes that it is unit, whereas it seems like the unit propagation process should be able to recognize this right away. This leads to a lot of redundant work not only the first time through the unit propagation of the assumptions, but every time the solver backtracks into or past the assumptions levels

(and then has to reconstruct those levels) as well. *One of the main contributions of this chapter is to introduce an extremely simple and light-weight technique for eliminating this redundancy.*

3.1.2 Summary of Our Contributions

There has been previous work on improving assumption-based SAT solving [61, 64, 65, 66]. However, in contrast to the techniques presented in other papers [64, 65, 66] the techniques we present in this chapter are much more light-weight. By this we mean two things: (1) our techniques are much easier to implement, e.g., no major new data-structures or algorithms are needed; and (2) our techniques yield good performance improvements on relatively easy instances that the SAT solver can solve in about two hundred seconds or less. In contrast, the heavy-weight techniques [64, 65, 66] are more complex to implement and the empirical evidence presented in these papers indicate that they often slow the SAT solver down on easier instances. These heavy-weight techniques do, however, often pay off (sometimes dramatically) on harder instances for which the SAT solver needs many hundreds or even thousands of seconds. So although our techniques continue to improve SAT solver performance on harder instances, the improvements they yield on such instances are unlikely to be as great as these heavy-weight techniques.

This is an important contrast as assumption-based SAT solving is applied in a diverse set of application areas. In model-checking applications the instances are often very large and very hard, and on these types of instances the heavy-weight techniques they describe can be very effective [65]. The application area we are most interested in, however, is MaxSAT solving. In that domain some solvers, e.g., MaxHS, use SAT solving to solve quite simple SAT instances [67], and the key to performance is SAT solver throughput, i.e., solving many instances quickly. Other MaxSAT solvers like RC2 [68] solve harder SAT instances than MaxHS, but most of these instances still take less than a few hundred seconds. We demonstrate that our techniques speed up both MaxHS and RC2, enabling both of these state-of-the-art solvers to solve more instances. Our techniques also speed up the MUS extraction tool Muser [16], showing that there is potential for this idea to work in other similar assumption-based SAT applications.

In particular, we present two light-weight techniques for speeding up assumption-based SAT solving. Our first technique is to enqueue all of the assumptions at once in one decision level, rather than the standard technique of sequentially making each assumption a decision and performing unit propagation after each decision. We show that the standard technique can suffer from unnecessary overhead that enqueueing all at once eliminates. We also provide an extensive empirical verification of the effectiveness of this simple idea. Our second technique is to develop a way to save and reuse the assumptions levels of the trail, both during the same SAT solving invocation and between different SAT solving invocations. Although our techniques are not technically sophisticated, they have a significant advantage in that they are very **cost-effective**: they are easy to implement and provide a non-trivial performance improvement.

3.1.3 Chapter Outline

In the rest of this chapter we will first give some necessary background. We will then motivate our first technique by demonstrating that the standard way of dealing with assumptions can incur overheads that are easily fixed by our approach. After describing some previous related work, we will then present our second technique which saves reusable parts of the trail when the assumptions are backtracked over. We will also show how "assumptions-level trail savings" can be realized between two SAT solving invocations that use

different sets of assumptions. Finally, we present empirical results that demonstrate that our techniques provide non-trivial performance improvements.

3.2 Background

We assume the reader is familiar with conflict-driven clause learning (CDCL) SAT solvers [69] and the concepts of unit propagation and conflict analysis. Please refer to Chapter 2 of this thesis and Chapter 4 of the Handbook of Satisfiability [70] for more detailed explanations. Below we will introduce some more specific notation and definitions that will be useful for the remainder of this chapter.

3.2.1 Conflict Clauses

When given an input CNF formula F, a SAT solver can produce either a satisfying truth assignment, or conclude that F is unsatisfiable. Assumption-based SAT solving extends the capacity of the SAT solver by asking it to solve F subject to a set of assumptions A which must be a set of literals. Now the SAT solver must either find a truth assignment satisfying $F \wedge A$ (i.e., a truth assignment satisfying F that also makes all of the literals in A true), or it must conclude that $F \wedge A$ is unsatisfiable. Furthermore, and most critical in many applications, when $F \wedge A$ is unsatisfiable the SAT solver must return a clause C such that (1) C contains only negated literals of A, i.e., $A \models \neg C$ and (2) $F \models C$. Putting (1) and (2) together we obtain $F \models \neg A$. We call any clause C that satisfies (1) and (2) a **conflict clause for** A. For any such clause C, every model of F must falsify at least one of the literals of A whose negation is contained in C.

It should be noted that the conflict clause C returned by the SAT solver need not be the smallest possible conflict clause. That is, there could be a smaller clause C', where $C' \not\subset C$, satisfying the above two conditions, but the SAT solver did not compute it. C is also not necessarily minimal, meaning there could be another clause $C'' \subset C$ which satisfies the above two conditions. In many applications the size of the returned conflict is important for performance: the shorter the returned conflict clause is, the more effective it tends to be for the application.

3.2.2 Unit Propagation

Modern SAT solvers use a two-watch-literal scheme [29] to effect efficient unit propagation. Treating a clause C as an indexed array of literals, the minisat scheme is to delegate the first two literals in the clause, $C[0]$ and $C[1]$, as the watchers. Associated with every literal ℓ is a list of all clauses that ℓ is currently a watcher for, $watchlist(\ell)$. Hence, for clause C, $C[0] = \ell$ or $C[1] = \ell$ if and only if $C \in watchlist(\ell)$.

CDCL SAT solvers utilize a trail containing the current path of the search tree being explored. This path consists of the set of literals currently assigned $true$. Let $val(\ell)$ denote the assigned truth value ($true$ or $false$) of literal ℓ. Hence, the trail contains a sequence of literals $\ell_1, \ldots, \ell_k$ such that $val(\ell_i) = true$ for $1 \leq i \leq k$. The literals on the trail are divided up into decision levels starting at zero. Decision level zero contains literals implied by the input formula F. Subsequent decision levels are started by finding a new unvalued literal d that the SAT solver decides to make $true$ (a decision literal). The trail's decision level is then incremented and that literal is then **enqueued**; i.e., it is assigned the value $true$ and it is added to the end of the trail. Whenever a literal is enqueued, its decision level is the trail's current decision level, so the decision literal d starts a new decision level in the trail.

The SAT solver keeps a unit-prop pointer to the last literal on the trail that has been unit propagated. The literals on the trail between the unit-prop pointer and the end of the trail have not yet been unit propagated. So whenever new literals are added to the trail, the suffix of un-propagated literals grows. Before the next decision is made, the SAT solver unit propagates every un-propagated literal on the trail moving the unit-prop pointer forward. This process might enqueue new literals, but it eventually terminates (by finding a conflict, or by running out of literals to unit propagate). After all literals have been unit propagated, the SAT solver starts a new decision level. Hence, all literals enqueued by unit propagation are at the same level as the most recent decision. If a conflict (a clause falsified by the trail) is found, unit propagation is terminated and the solver backtracks after learning a new clause. If all variables have been valued, the SAT solver terminates and returns "satisfiable".

The process of unit propagating a literal ℓ involves examining all clauses in $watchlist(\neg\ell)$ to determine if any of them have become falsified or unit (i.e., all but one literal in the clause has been falsified). A clause cannot become unit unless one of its watches has become false, hence it is sufficient to check only the clauses in $watchlist(\neg\ell)$ when ℓ is made $true$. If the clause $C \in watchlist(\neg\ell)$ has become falsified, then unit propagation can stop and clause learning and backtracking can occur. If C has become unit with sole remaining unfalsified literal x, then x can be enqueued if it is not already on the trail. Determining if C has become falsified or unit requires examining the non-watcher literals in C (i.e., from $C[2]$ onwards) until a non-false non-watcher literal x is found. In the worst case this requires examining all literals in C (except for $C[0]$ and $C[1]$). If such a literal x is found, it replaces ℓ as one of C's watches. That is, x and ℓ are swapped with each other in C, and C is removed from $watchlist(\neg\ell)$ and placed on $watchlist(\neg x)$. If no such x exists, then if C's other watch is unvalued that makes it the sole remaining non-false literal in C and it can be enqueued; otherwise if C's other watch is already false, C has become falsified.[1]

3.3 Enqueueing all Assumptions at once

In this section, we will introduce the main technique we are contributing in this chapter and show a side-by-side comparison of how it works against how the standard approach works.

3.3.1 Standard Approach

Let the input formula be F and the assumptions $A = \{a_1, \ldots, a_k\}$. The standard technique of supporting assumptions [46] used in most modern SAT solvers is to require that the SAT solver choose a_i as the decision literal whenever it starts decision level i for $1 \leq i \leq k$. If a_i is already $true$ the SAT solver increments the decision level[2] and continues to the $i{+}1$-th decision, i.e., an empty decision level is added to the trail. If a_i is already $false$, then it is the case that the previous $i{-}1$ assumption decisions were sufficient to imply $\neg a_i$. So $C = (\neg a_1, \ldots, \neg a_i)$ is already a clause such that $F \models (\neg a_1, \ldots, \neg a_i)$ and $A \models \neg C$. That is, C is a conflict clause for A. However, by conflict analysis (described in more detail below) a shorter conflict than C can typically be computed. Otherwise a_i is still unvalued, and it is enqueued as a new decision. After all assumptions are enqueued and unit propagated, the SAT solver is free to make decisions according to its normal heuristics.

[1] Quick checks to determine if C is already satisfied can be made first by checking data in the watch data structure and checking if the clause's other watch is $true$.

[2] Incrementing the decision level for assumptions that are already true is for bookkeeping purposes only, so that the decision level at the end of deciding on all assumptions is equal to one more than the total number of assumptions.

Since A always becomes a prefix of the trail, if a satisfying model is found then that model must satisfy $F \wedge A$. Otherwise, eventually a conflict will be found that forces $\neg a_j$ for some j at some level i where $i < j$. In that case the SAT solver will backtrack to level i and add $\neg a_j$ as a new unit implicant. Then when the SAT solver descends again to level j, it will find that a_j is already $false$ and will invoke conflict analysis to compute and return a conflict clause over A. Note that irrespective of the satisfiability status of $F \wedge A$, the SAT solver during its search might learn any number of new clauses that cause new unit implicants to be added into the assumption levels (levels $i \leq k$). Each such new implicant at level i will cause the SAT solver to undo the decisions $a_{i+1}, \ldots, a_k$, after which it will have to make those decisions again as it re-descends the search tree.

This backtracking into the various assumption levels to add new implied literals and then having to redo the remaining assumption decisions is the **first inefficiency** of the standard technique. In some applications, particularly in MaxSAT solving, there can often be thousands of assumptions, so this inefficiency can have a significant impact. Example 1 shows that this inefficiency can potentially induce an overhead that is quadratic in the number of assumptions.

Example 1 *Let the assumptions $A = \{a_1, \ldots, a_n\}$ be processed by the SAT solver in this order, and the input formula F consist of three sets of clauses: (1) $\{(\neg a_1, z_i^1, z_i^2) | i = 1 \ldots n\}$, (2) $\{(\neg y_i, \neg z_i^1) | i = 1 \ldots n\}$, and (3) $\{(z_i^1, \neg z_i^2) | i = 1 \ldots n\}$. Furthermore, assume that after setting the assumptions, the SAT solver's branching heuristic selects variables $y_1, \ldots, y_n$ in that order before any of the variables $z_1^1, z_1^2, \ldots, z_n^1, z_n^2$, and follows a default phase of first setting each literal to $true$.*

On its first descent the SAT solver will assume $a_1, \ldots, a_n$ in the first n decision levels. None of these decisions cause any unit propagations. Then it will select y_1 as its $n{+}1$-th decision. The literal $\neg z_1^1$ will be implied by unit propagation from the clause $(\neg y_1, \neg z_1^1)$ in set (2). Then the literal $\neg z_1^2$ will be implied from the clause $(z_1^1, \neg z_1^2)$ in set (3). This will falsify the clause $(\neg a_1, z_1^1, z_1^2)$ in set (1). The 1-UIP clause $(\neg a_1, z_1^1)$ will be generated from resolving the conflict $(\neg a_1, z_1^1, z_1^2)$ against the reason clause $(z_1^1, \neg z_1^2)$ for $\neg z_1^2$. This will cause the SAT solver to backtrack to level 1, where both z_1^1 and $\neg y_1$ will be unit implied.

After this the SAT solver will repeat setting $a_2, \ldots, a_n$ as assumptions, and at decision level $n{+}1$ will select y_2 as the decision literal. Applying the same reasoning as before, except with the clauses $(\neg y_2, \neg z_2^1)$, $(z_2^1, \neg z_1^2)$, and $(\neg a_1, z_2^1, z_2^2)$, we see that again the SAT solver will backtrack to level 1 where it will add the unit implicants z_2^1 and $\neg y_2$. This would continue to happen n times, giving rise to the SAT solver making the assumption decisions $O(n^2)$ times before concluding that all clauses of F are satisfied. ■

The **second inefficiency** of the standard technique arises from the observation by Gent [71] that the unit propagation scheme used in most modern CDCL solvers can visit the literals of the **same** clause $O(n^2)$ times in the worst case, where n is the length of the clause, when descending a single branch.[3] This second inefficiency arises from having to move a clause from one watch list to another $O(n)$ times and each time having to scan $O(n)$ literals in the clause, as shown in Example 2.

Example 2 *Let the assumptions $A = \{a_1, \ldots, a_n\}$ be processed by the SAT solver in this order. Consider the length n clause $C = (x, \neg a_1, \neg a_2, \ldots, \neg a_{n-1})$. The two watchers are x (at $C[0]$) and a_1 (at $C[1]$). The assumption a_1 is enqueued first by the SAT solver, and when unit propagated $C \in watchlist(\neg a_1)$ will be checked from $C[2]$ onwards for a new non-false non-watcher. This search will find $\neg a_2$ at position $C[2]$ and make it a new watch by swapping $\neg a_1$ (at $C[1]$) and $\neg a_2$ (at $C[2]$) and placing C in $watchlist(\neg a_2)$. The*

[3] Gent's solution to this inefficiency will be discussed later.

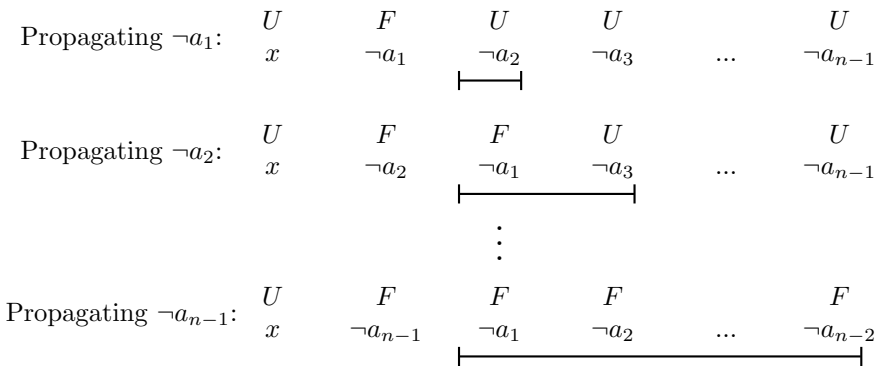

Figure 3.1: The propagate routine searching for a new watcher for clause C of Example 2 when each assumptions is made at a separate decision level and unit propagated before moving to the next assumption. The first two literals are the current watcher literals and the line shows the span of literals traversed in searching for a replacement watcher. Truth values are above each literal; "F" represents false, "T" represents true, and "U" represents unassigned.

assumption a_2 will be enqueued next, and $C \in watchlist(\neg a_2)$ will be searched again, this time from $C[2]$ to $C[3]$, before a new watch, $\neg a_3$ is found at $C[3]$. Literals $\neg a_2$ (at $C[1]$) and $\neg a_3$ (at $C[3]$) will be swapped and C placed in $watchlist(\neg a_3)$. When the i-th assumption a_i is made, C will be on $watchlist(\neg a_i)$, and positions $C[2]$ to $C[i+1]$ will be searched until finding $\neg a_{i+1}$ as a new watch at position $C[i+1]$. Literals $\neg a_i$ (at $C[1]$) and $\neg a_{i+1}$ (at $C[i+1]$) will be swapped and C placed in $watchlist(\neg a_{i+1})$. In total, in making the first n assumptions the SAT solver will have to visit $O(n^2)$ literals in C to finally conclude that x is unit implied by the assumptions.[4] *Figure 3.1 illustrates this process.* ■

3.3.2 New Approach

The technique we propose for fixing both of these inefficiencies is very simple. **After all literals at level zero have been unit propagated, the SAT solver increments the decision level and enqueues all assumptions at decision level 1, after which it performs unit propagation.** So all assumptions are placed on the trail at the top of level 1, and unit propagation is performed only after all assumption literals are on the trail and have been assigned the value $true$.

If, when processing the assumptions, the SAT solver finds one a_i that is already $true$, then a_i must have been made true at level 0 since unit propagation has not yet been run at level 1. That is, a_i is already on the trail in level 0 and we do not need to enqueue it again, so we can skip it. Similarly, if a_i is already $false$ then $\neg a_i$ must have been made true at level 0, so $F \models (\neg a_i)$ and the SAT solver can return the conflict clause $(\neg a_i)$.

Enqueueing all assumptions at level 1 also necessitates a change to the SAT solver's **analyzeFinal** routine, which is called in the standard technique to compute a conflict clause when an assumption a_i is about to be enqueued as the i-th decision and is discovered to be $false$. We will describe those changes below, but first we explain how our technique fixes the two inefficiencies identified above.

With our technique all assumptions are at level 1, so if a new unit implicant of the assumptions is found during search, that clause will cause a backtrack to level 1. None of the assignments in level 1 will be

[4] This description follows the minisat and Glucose schemes for watchers, but this particular type of implementation is not necessary. Scanning $O(n^2)$ literals in the clause down a single branch occurs with any implementation that stores no information about the previous scan [71].

Propagating all	U	F	F	F		F
assumptions at once:	x	$\neg a_1$	$\neg a_2$	$\neg a_3$	...	$\neg a_{n-1}$

Figure 3.2: The propagate routine searching for a new watcher when all assumptions are enqueued before unit propagation on the clause from Example 2. Line shows span of literals traversed in searching for a replacement watch.

undone, instead the new unit literal will be added to the bottom of level 1 and search will continue after unit propagation is run on the new unit literal. This resolves the first inefficiency.

Example 3 (Example 1 continued) *With the same set of assumptions A and input formula F as Example 1 our technique would operate as follows. First all assumption literals $a_1, \ldots a_n$ would be enqueued at level 1 (in this example F has no initial unit clauses so level 0 will be empty). As before, unit propagation will not find any implied units. Then the SAT solver will increment the decision level to 2 and select y_1 as the next decision. Unit propagation operates in the same way as in Example 1, and the learnt 1-UIP clause will again be $(\neg a_1, z_1^1)$. This will cause the SAT solver to backtrack to level 1, where it will add z_1^1 and $\neg y_1$ at the bottom of level 1 without disturbing any of the assumption literals on the trail. The SAT solver will then make a new decision, y_2, and in the same way a backtrack to level 1 will be generated where the new units z_2^1 and $\neg y_2$ will be added at the bottom of level 1 without disturbing the previously added units z_1^1 and $\neg y_1$. This will continue n times until all y_i are set and all clauses are satisfied. So this process will require making only n instead of $O(n^2)$ assumption decisions.* ■

Our technique also addresses the second inefficiency. In particular, since all assumption literals are valued before unit propagation starts, no clause will ever be moved to the watch list of a negated assumption literal as all of these literals are already $false$.

Example 4 (Example 2 continued) *With the same set of assumptions A and clause $C = (x, \neg a_1, \neg a_2, \ldots, \neg a_{n-1})$ as Example 2, our technique would operate as shown in Figure 3.2. Since all of the assumption literals in the clause have already been made false, unit propagation will visit the literals from $C[2]$ to $C[n-1]$ only once, concluding that x is unit implied by the assumptions. That is, with our technique only $O(n)$ literals of C will be examined instead of $O(n^2)$ literals with the standard technique.* ■

3.3.3 Implementation

Our new approach of enqueueing all assumptions at once is very simple to implement, and we illustrate this in the framework of the minisat code base. It should be equally easy to implement our approach in non-minisat based SAT solvers. Two routines need to be altered: (1) the main **search()** routine that makes new decisions, invokes unit propagation, and performs clause learning when a conflict is detected; and (2) the **analyzeFinal()** routine that computes the final conflict clause. The needed changes to the **search()** routine are shown in Algorithm 1. If a conflict occurs at decision level 1 (where the assumptions are enqueued) we must convert it into a conflict for the assumptions (line 6); and if we are making a decision at level 0 we enqueue all assumptions at level 1 (lines 19–26).

The routine **analyzeFinal()** also changes. In the standard approach it is passed an assumption literal that has been falsified by a prior set of assumption decisions (line 13), whereas in the new approach it is passed the conflict clause found at level 1 by unit propagation (line 6). In both cases the routine must return the computed conflict in the passed **conflict** vector.

Algorithm 1: Code changes for a minisat based SAT Solver to implement enqueueing all assumptions at once. Line 6 is added and lines 11–17 are replaced with lines 19–26.

```
1  search ()
2  while true do
3      confl = unitPropagate()
4      if confl then
5          if decisionLevel() ≡ 0 then return false;
6          if decisionLevel() ≡ 1 then                                  /* ADD THIS LINE*/
7              analyzeFinal(confl, conflict); return false;
8          analyze conflict and backtrack
9      else                                                             /* No conflict */
10         begin /* REMOVE THIS BLOCK /*
11             while assumptions remain do
12                 a_i = nextAssumption();
13                 if value(a_i) ≡ false then analyzeFinal(¬a_i, conflict);
14                 else if value(a_i) ≡ true then newDecisionLevel();
15                 else nextDecision = a_i; break;
16             if nextDecision ≡ NIL then nextDecision = heuristic();
17             newDecisionLevel(); enqueue(nextDecision);
18         begin /* ADD THIS BLOCK */
19             if decisionLevel() == 0 then
20                 newDecisonLevel()
21                 forall Assumptions a_i do
22                     if value(a_i) ≡ false then conflict = {a_i}; return false;
23                     if value(a_i) ≢ true then enqueue(a_i);
24             else
25                 nextDecision = heuristic()
26                 newDecisionLevel(); enqueue(nextDecision)
```

The standard minisat implementation starts with C equal to $\neg a_i$'s reason clause (i.e., the clause that became unit implying $\neg a_i$). Then, while there exists a literal ℓ in C not equal to $\neg a_i$ and such that $\neg\ell$ has been unit implied by reason clause $reason(\neg\ell)$, we replace C by the resolution of C and $reason(\neg\ell)$. The end result is a conflict clause that contains $\neg a_i$ and the negation of decision literals. Since, $\neg a_i$ was implied at a level where all decisions are assumptions, the computed clause contains only negated assumption literals as required.

The new implementation is very similar. It starts with C equal to the passed conflict. Then, while there exists a literal ℓ in C such that $\neg\ell$ has been unit implied by reason clause $reason(\neg\ell)$, we replace C by the resolution of C and $reason(\neg\ell)$. Since the conflict occurs at level 1, only the assumption literals have no reason clause, so this process removes all non-assumption literals from C and the final result contains only negated assumption literals as required.

It should be noted however that these two different approaches can produce different conflict clauses even if the initial conflict and trail are similar. In some cases the standard approach might produce a shorter clause and in other cases our new approach might produce a shorter clause. The following example illustrates each of these cases.

Example 5 *The following table illustrates a case where the standard technique will learn a shorter conflict clause.*

Standard			*Enqueue All*		
Level	*Lit*	*Reason*	*Level*	*Lit*	*Reason*
1	a_1	*nil*	*1*	a_1	*nil*
1	a_2	$(a_2, \neg a_1)$	*1*	a_2	*nil*
1	a_3	$(a_3, \neg a_1)$	*1*	a_3	*nil*
1	$\neg a_4$	$(\neg a_4, \neg a_3, \neg a_2, \neg a_1)$	*1*	a_4	*nil*
2	*empty level*		*Conflict at level 1:*		
3	*empty level*		$(\neg a_4, \neg a_3, \neg a_2, \neg a_1)$		
4	*Conflict at level 4:* $\neg a_4$				

The table shows the trail when a conflict is found. The left three columns show the trail for the standard approach, and the right three show the trail for our proposed approach of enqueueing all assumptions at level 1. The table indicates the decision level, the literal on the trail and the reason clause. Literals with non-nil reasons are implied literals.

The standard approach detects a conflict at level 4 when it tries to make a_4 true as the next decision and finds that a_4 is already false. The conflict it learns starts with the reason clause for $\neg a_4$, $(\neg a_4, \neg a_3, \neg a_2, \neg a_1)$, and proceeds to resolve away all implied literals from this clause except for $\neg a_4$. This involves resolving away a_3 and a_2 to obtain the conflict $(\neg a_4, \neg a_1)$.

Our new technique, on the other hand, will enqueue all assumptions at level 1, and then discover that the clause $(\neg a_4, \neg a_3, \neg a_2, \neg a_1)$ is falsified. In the new technique none of these literals are implied, so no resolution steps are performed. Hence it will return this longer clause as the conflict.

On the other hand, the following table illustrates a case where the standard technique learns a longer conflict clause.

Standard			*Enqueue All*		
Level	*Lit*	*Reason*	*Level*	*Lit*	*Reason*
1	a_1	*nil*	*1*	a_1	*nil*
2	a_2	*nil*	*1*	a_2	*nil*
3	a_3	*nil*	*1*	a_3	*nil*
3	a_4	$(a_4, \neg a_3, \neg a_2, \neg a_1)$	*1*	a_4	*nil*
3	$\neg a_5$	$(\neg a_5, \neg a_4)$	*1*	a_5	*nil*
4	*empty level*		*Conflict at level 1:*		
5	*Conflict at level 5:* $\neg a_5$		$(\neg a_5, \neg a_4)$		

The standard technique discovers a conflict at level 5 when it finds that a_5 is already falsified. Starting with the reason clause $(\neg a_5, \neg a_4)$, it will resolve away the implied literal $\neg a_4$ to obtain the conflict clause $(\neg a_5, \neg a_3, \neg a_2, \neg a_1)$.

Our new technique, on the other hand, will return the shorter clause $(\neg a_5, \neg a_4)$ as the conflict, as again none of these literals are implied so no resolution steps will be performed. ■

As we will demonstrate later in our experiments, although our technique could return longer conflicts in the same context, it generally returns shorter ones than the standard technique.

3.3.4 Previous Work

Approaches that do not address the first inefficiency

Gent [71] proposed an alternate method for eliminating the second inefficiency where a clause moves from the watch list of one assumption to another many times and each time has many of its literals scanned. In

particular, he proposed keeping track of, for each clause, the location where the previous search for a non-falsified literal stopped. Then when a new search is made, the search starts at this previous location and, if necessary, wraps around. Gent showed that this reduces the worst case number of literals that could be visited in a single clause along a single branch to $2n$ down from $O(n^2)$ (n is the clause length). Unlike our approach, Gent's method is more intrusive, requiring a change to the clause data structure (to track the location where the previous search stopped). However, it accounts for non-assumption literals as well as assumption literals. Nevertheless, Gent also showed that his technique did not yield any significant gains in SAT solver performance on the instances he experimented with, whereas our technique does yield performance gains, perhaps because it fixes both inefficiencies.

Audemard et al. [61] proposed four lightweight techniques for improving assumption-based SAT solving in Glucose. First, they ignored the assumption literals when computing the LBD score (refer to Section 2.1.1 for the definition of LBD) of a learnt clause. Our method comes close to achieving the same improvement: with it all assumption literals are at the same level, so they contribute only 1 (instead of 0) to the LBD score. Second, they store all of the assumption literals at the end of each learnt clause so as to avoid having to scan these literals on LBD score updates. Third, they avoid making assumption literals become watcher literals for clauses, where possible, so that when an assumption is inevitably made it will have fewer clauses on its watch list to scan through.[5] Our technique does not achieve the second nor the third improvement.

Fourth, during unit propagation when scanning a clause for a new non-false watcher, they continue searching until they find a non-false non-assumption literal. If none exist they use the last non-false literal found (even if it is an assumption literal). This technique addresses some of the second inefficiency, but not all of it. In particular, if in Example 2 the assumptions A are set in the order a_1, a_{n-1}, a_{n-2}, ..., a_2 then the clause $(x, \neg a_1, \neg a_2, \ldots, \neg a_{n-1})$ will still have its literals scanned $O(n^2)$ times when each assumption is a separate decision. For example, when a_1 is made true, all of the literals $\neg a_2$, ..., $\neg a_{n-1}$ will be scanned, skipping over non-false assumption literals, until finally returning the last non-false assumption literal $\neg a_{n-1}$ as the new watcher.

It can also be noted that none of the techniques by Audemard et al. [61] or Gent [71] address the first inefficiency.

Approaches that address the first inefficiency but incur additional overhead

Lagniez et al. [64] propose a technique to replace all assumption literals in a learnt clause with a fresh literal, referred to as an abbreviation. This abbreviation is defined to be equivalent to the disjunction of the negation of assumptions it is replacing, and this is stored in a new data structure maintained by the solver called a definition map. Their approach ends up enqueueing all assumptions at once similar to our approach, but also involves scanning through the definition map which is very costly. Our attempts at implementing abbreviations in MaxSAT solvers did not yield positive results. Aside from the abbreviations, another key difference between our technique and theirs is that we are allowing the assumptions to become unassigned during solving whenever the solver backtracks to level 0, and we found this to be beneficial.

Other approaches proposed by Nadel et al. [65, 66] add all of the assumptions as unit clauses, so that full preprocessing can take place, and then examine the resolution refutation at the end of the solving invocation to determine all assumptions that each learnt clause used in its derivation. Then to reuse learnt clauses from one invocation in subsequent invocations, one must do post-processing on the learnt clauses with the resolution refutation proof in order to make them persistent (alternatively referred to as "pervasive"). Although this

[5] It is not clear if this third technique is an improvement outside of the context of MUS extraction.

can address the inefficiencies we have discussed (by getting rid of assumptions altogether for the remainder of each solving invocation), the overhead of maintaining a resolution refutation proof and traversing it for potentially every learnt clause means this technique is unlikely to be effective in the context of MaxSAT, which relies on fast solver throughput.

3.4 Assumptions-Level Trail Savings

In this section, we will introduce a second technique we are contributing in this chapter, which was still beneficial to the solver, but less so than the technique presented in Section 3.3 was.

3.4.1 Motivating Example

When a restart returns the SAT solver to level zero, the solver can often proceed to reproduce the same initial sequence of decisions that were on the trail before the restart. Redoing these decisions is redundant work and methods for saving this work, by making the SAT solver backtrack only to the point where the restart causes a divergence in the decisions made, have been developed [72]. Similarly, the paper by Audemard et al. [61] proposes only backtracking to the bottom of the assumption levels on restart, as all of the assumption decisions will be redone by the solver. However, these techniques do not help when the SAT solver learns a new unit clause. Unit clauses are added to level 0, so the solver must backtrack across the assumptions to insert the new unit literal into the trail. After this, it must redo the assumptions, as in the example below.

Example 6 *Let F be the formula, let the assumptions $A = \{a_1, \ldots, a_k, \ldots, a_n\}$ be processed by the SAT solver in this order, and let $P = \{p_1, \ldots, p_n\}$ be all of the other literals on the trail at level 1 (i.e., the unit propagants of $F \wedge A$). Let $C_l = l$ be a unit clause the solver has just learned and assume that unit propagating $F \wedge A \wedge C_l$ does not cause a contradiction.*

The solver will backtrack back to the end of level 0, unit propagate l, then enqueue and unit propagate all literals in A again. The new set of unit propagants that will appear on the trail is guaranteed to be a superset of P, thus repeating the work of the previous solve's propagation of A. ■

3.4.2 New Approach

Our assumptions-level trail savings method is based on the observation that after a new unit is added to level 0, the set of literals forced to be true at level 0 and level 1 can only either (a) be contradictory or (b) be a superset of the previous set of literals forced at level 0 and level 1. In particular, let L_0 and L_1 be the literals at level 0 and level 1 before a new unit clause (ℓ) is found. The new unit clause will cause a backtrack to the end of level 0 where ℓ will be added and unit propagated. Let L'_0 be the new literals at level 0 after this process. If no conflict is found we have that $L_0 \subset L'_0$. Let L'_1 be the new level 1 generated by enqueueing all the assumptions not already in L'_0 and then performing unit propagation. If L'_1 does not generate a contradiction, we must have that $L_1 \subseteq L'_0 \cup L'_1$. If $x \in L_1$ is an assumption, then $x \in L'_0 \cup L'_1$ as L'_1 starts with all assumptions not already at level 0. Considering the unit implicants $x \in L_1$ in the order they appeared in the trail, we can conclude inductively that for every other literal l in x's reason clause we have that $\neg l \in L'_0 \cup L'_1$. Hence, x must also be a unit implicant at level 1 or 0. Therefore, $L_1 \subseteq L'_0 \cup L'_1$.

Our new technique based on this observation is as follows. When a new unit is learnt, we save a copy of the level 1 trail (all literals and clause reasons). Then we backtrack the trail to the end of level 0, add the

new unit, and perform unit propagation. If no contradiction is found, we then enqueue each literal from our copy of the level 1, preserving the order these literals previously appeared on the trail (so assumptions are enqueued first). If any of these literals is already $true$, we skip enqueueing it. If any of these literals is already $false$, we can compute a conflict for the assumptions. If the falsified literal is an assumption $\neg a_i$ then the conflict is $(\neg a_i)$, otherwise the conflict is computed by passing the stored reason clause associated with the falsified literal (this reason clause has become falsified at level 1) to **analyzeFinal** to compute the conflict. After all saved literals have been added to level 1, we invoke unit propagation from the beginning of level 1. Note that although we can restore the literals from level 1, we still have to unit propagate them once again. However, this unit propagation process is sped up by the fact that many literals have already been made $true$ at level 1. Hence, the second inefficiency of moving a clause from one watcher to another will occur less frequently. This is illustrated by contrasting Example 6 with Example 7.

Example 7 *[Example 6 continued] With the same formula, set of assumptions, unit propagants, and learnt unit clause as Example 6, assume now that we are using assumptions-level trail saving. This means upon learning C_l, we save a copy of every literal on the trail at level 1 (which is $A \cup P$), along with their corresponding reasons, before backtracking to the end of level 0.*

The solver will then backtrack back to the end of level 0, unit propagate l, then enqueue and unit propagate all literals in $A \cup P$ at once (since we assumed that $F \wedge A \wedge C_l$ does not cause a contradiction by unit propagation). Hence, the work from the previous solve's propagation of A required to get all literals in P does not need to be re-done. ■

Remark. It is worth noting that one does not have to backtrack past the assumptions at all when learning a unit clause, which could theoretically save even more work than the method described in this section. To illustrate this point, consider a solver which learns a unit clause (l) and finds two other level 0 propagants, l_1 and l_2, after enqueueing and propagating l. It is perfectly sound to instead backtrack to level 1, enqueue l and mark it specially as part of level 0, and then propagate. Propagating will still find the propagants l_1 and l_2, but will mark them as part of level 1. At the end of the solving invocation upon backtracking to level 0, all of the literals marked as level 0 (including l in this example) can be added to the solver. One last propagation after adding these literals will recover all of the level 0 literals which are implied by unit propagation (including l_1 and l_2 in this example).

However, we have observed empirically that backtracking to level 0 on learnt units performed better than backtracking to level 1 in this way. We hypothesize that this could be from the fact that the solver can choose fresh reasons for some of the level 1 propagants and implicants when it is allowed to backtrack to level 0.

Using the same setup as in Section 7 and with a large set of 7439 benchmarks that includes all instances from the MaxSAT Evaluation from 2006 onward, we modified the MaxHS version which propagates the set of assumptions together to also only backtrack to level 1 on learnt units, store those learnt units in a vector, and supply those learnt units to level 0 of the solver at the end of the solving invocation. This resulted in the total number of instances solved decreasing from 6131 to 6123. If the learnt units are not stored and "forgotten" at the end of the solving invocation, it further decreased the total number of instances solved to 6093.

3.5 Propagating the final core

We were able to minimize the core by a slight amount (and also slightly improve the MaxSAT solver performance) in some cases by taking the final core, which consists of negated assumptions, and unit propagating each of them in order. Because we enqueued the assumptions together as a set, it is possible for assumptions in the final core to be implied by other literals in the final core by unit propagation in the order they appear on the trail. This wouldn't happen with the standard technique because each assumption is unit propagated one at a time, so any assumptions that are implied by previous assumptions become propagants and can not be in the final conflict clause (and thus can not appear in the final core).

3.5.1 Interaction with Abstract Cores

Since the publication of our ideas, another idea called abstract cores [73] was introduced which substantially improved the performance of MaxHS. Abstract cores are a compact representation of potentially exponentially many cores by accounting for cardinality constraints in the representation of the cores. The exact details are beyond the scope of this thesis, but are very interesting and can be found in the paper by Berg et al [73].

Unfortunately, enqueueing all assumptions at once does not interact with abstract cores very well, and can even cause the overall performance of MaxHS to decrease slightly (although not usually by a significant amount). The exact reason for this would require more research and is an interesting future direction, but we hypothesize that it may have to do with the order in which the assumptions are unit propagated.

One of the main reasons using abstract cores was so successful, was because they improved the performance of MaxHS on DRMX-AtMostK [74] instances dramatically. Without abstract cores, almost none of the DRMX-AtMostK instances can be solved by MaxHS, and with abstract cores all of them can be solved.

One solution we proposed is to allow the SAT solver to use our technique of enqueueing all assumptions at once throughout its entire search, and instead of taking the final conflict clause that we naturally find at the assumptions levels, we instead just go back and unit propagate all assumptions again in order (not just assumptions occurring in the conflict clause). Because we know that the solver will definitely find a conflict now (namely, the one we just naturally found), we are guaranteed to get a conflict again before leaving the assumptions levels, but this time we might get a different conflict with literals that are closer together in their natural encoding order. This allowed MaxHS (using abstract cores) to once again solve all of the DRMX-AtMostK instances in the 2018 competition. We found, however, that this does not make a significant difference in performance on MaxHS overall on a large set of benchmarks. We think there is still potential to better incorporate our technique of enqueueing all assumptions at once with abstract cores, but that is left to future work.

3.6 Assumptions-Level Trail Savings Between SAT invocations

In this section we will show an extension of the technique in Section 3.4 that allows one to save some literals in between SAT invocations, instead of just saving literals within the same invocation.

3.6.1 Extending Assumptions-Level Trail Savings

We can further extend assumptions-level trail savings to provide a head start for the SAT solver when it is called again with a different set of assumptions. Note that level 0 is always preserved between calls to the

SAT solver as this level does not depend on the assumptions. Our extension is to also try to preserve as much of level 1 as is possible between SAT calls. The method is to save all of the literals implied at level 1, and their reason clauses, at the time the SAT solver exits. Then when the SAT solver is called again with a new set of assumptions, we enqueue all of these assumptions at level 1 as normal. After the new assumptions are enqueued, and before they are unit propagated, we check all of the saved literals previously implied at level 1, in the order they were previously on the trail. If their reasons are still unit clauses under the new trail, we enqueue them on the trail with the same reason clause.[6] Note that by examining these implied literals in trail order we can detect preserved implied literals that rely on previously preserved implied literals. For example, say x and y were at level 1 at the end of the previous SAT solve with y appearing after x, and with reason clauses $(x, \neg a_1)$ and $(y, \neg x)$. Then if the new SAT call includes the previous assumption a_1 our technique will detect that x is still unit implied with the same reason clause, and x will be added to the trail. Then y will also be detected to still be unit because x has already been added to the trail. Example 8 illustrates how this extension to assumption-levels trail saving can be effective.

Example 8 *Suppose there are two SAT solving invocations which come one after another. Let the assumptions be $A_1 = \{a_1, a_2, a_3, a_4\}$, processed by the SAT solver in this order, for the first solving invocation and let the assumptions be $A_2 = \{a_1, a_5, a_3, a_6\}$, processed by the SAT solver in this order, for the second solving invocation. Let the literals $\{p_1, p_2, p_3\}$ be the unit propagants of A_1 at the end of the first solving invocation and let the literals $\{p_1, p_3\}$ be the unit propagants after unit propagating A_2 in the second solving invocation. Let the clauses $\{r_1 = \{p_1, \neg a_1\}, r_2 = \{p_2, \neg a_1, \neg a_2\}, r_3 = \{p_1, \neg a_1, \neg a_3\}\}$ be the corresponding reasons such that for all i, the reason of p_i is r_i.*

Using the extension to assumptions-level trail saving that saves the level 1 in between SAT invocations, we make a copy of $\{p_1, p_2, p_3\}$ along with $\{r_1, r_2, r_3\}$ after the first SAT solving invocation. At the beginning of the second SAT solving invocation, we enqueue the relevant assumptions A_2 and also scan clauses $\{r_1, r_2, r_3\}$ to check which are still unit. In our example, r_1 and r_3 are still unit, meaning we can enqueue p_1 and p_3 along with the assumptions before doing unit propagation. By the same principle shown in previous examples, enqueueing these unit propagants up front can help save time scanning watch lists during unit propagation. The reason clauses $\{r_1, r_2, r_3\}$ would likely have had to be scanned at least once during the unit propagation process anyway, so we are unlikely to lose time from scanning useless clauses. ■

3.7 Experiments and Results

In this section, we will first describe the setup in Section 3.7.1 and benchmarks used in the experiments we ran before getting to the results. The first experiments presented in Section 3.7.2 were designed to test whether our techniques sped up and solved more instances for MaxSAT, and are the most important experiments of the chapter. They indeed confirm that our techniques were successful.

The next experiment in Section 3.7.3 was a finer-grained experiment, designed to check whether the actual SAT solver within the MaxSAT solvers was really speeding up most of the time. After confirming that was the case, we designed an experiment in Section 3.7.4 to compare the average core sizes produced by our enqueueing technique with the average core sizes produced by the standard approach, which surprisingly showed our technique to be producing the smaller cores. This is significant because it means that our positive

[6] We save a reference to the reason clause, so before checking to see if the reason is still unit we must ensure that the references haven't been changed by garbage collection, and that the implied literal is still at position zero in the reason clause (this is a minisat invariant for reason clauses).

results are not only coming from a speedup in the SAT solver, but likely also coming from our technique developing smaller cores and benefiting the MaxSAT algorithm.

Lastly, we show some experiments in Section 3.7.5 confirming that our techniques do speed up another assumption-based SAT application, MUS extraction, but do not solve more instances.

3.7.1 Setup and Benchmarks

We implemented our techniques in the minisat 2.2 and Glucose 3.0 SAT solvers. In Glucose 3.0 we also preserved the already implemented incremental techniques of the Audemard et al. paper [61]. We then used these modified SAT solvers in the MaxHS [67] and RC2 [68] MaxSAT solvers, both of which are state-of-the-art MaxSAT solvers. MaxHS uses minisat while RC2 uses Glucose.

We then ran these solvers using both the modified and original SAT solvers on 7439 benchmark instances (4627 unweighted and 2812 weighted) collected from the 2008 to 2018 MaxSAT Evaluations [75]. This benchmark set includes all non-random instances used and submitted to these evaluations, excluding 825 "abrame-habet" random instances that were categorized as "crafted" instances (none of these are solvable by either MaxHS nor RC2). We also removed duplicate instances that had different names but were the same except for comment lines. The experiments were run on 2.4GHz Intel cores with 30min CPU time and 5.24GB memory limits.

3.7.2 Main Experiments for MaxSAT

	Total				Unweighted		Weighted	
	MaxHS	+/-	RC2	+/-	MaxHS	RC2	MaxHS	RC2
original	6052	0/0	6030	0/0	3940	3993	2112	2037
enqueue assumptions as set	6131	114/35	6081	99/48	3995	4032	2136	2049
enqueue assumptions as set + assumptions-based trail saving	6136	120/36	6079	95/46	3991	4030	2145	2049
enqueue assumptions as set + assumptions-based trail saving + save literals from last invocation	6138	115/29	6080	92/42	3989	4030	2149	2050

Figure 3.3: Number of MaxSAT instances solved by MaxHS and RC2 using different extensions of the underlying SAT solver. The enqueueing assumptions technique allows MaxHS to solve almost 80 more instances and allows RC2 to solve more than 50 more instances. The trail saving techniques have a modest but positive impact for MaxHS, but not much of an impact for RC2. For reference, in the "+/-" column, the first number shows the number of instances that solver solved but the original solver did not (instances gained), and the second number represents the total number of instances that solver did not solve but the original solver did (instances lost).

Our first set of experiments were simply comparing how each of our techniques impacted two state-of-the-art MaxSAT solvers. These are the main results of this chapter that tell us that these techniques were successful.

Figure 3.3 shows that our first technique of enqueueing all assumptions at once is surprisingly effective yielding 79 newly solved instances for MaxHS and 51 newly solved instances for RC2. It should be noted that both of these solvers are state-of-the-art and techniques that allow them to solve this many new instances are not easy to find. Assumptions-level trail savings within the same SAT solver call is a less successful improvement. It gains 5 more problems for MaxHS but loses 2 problems for RC2. Assumptions-level trail

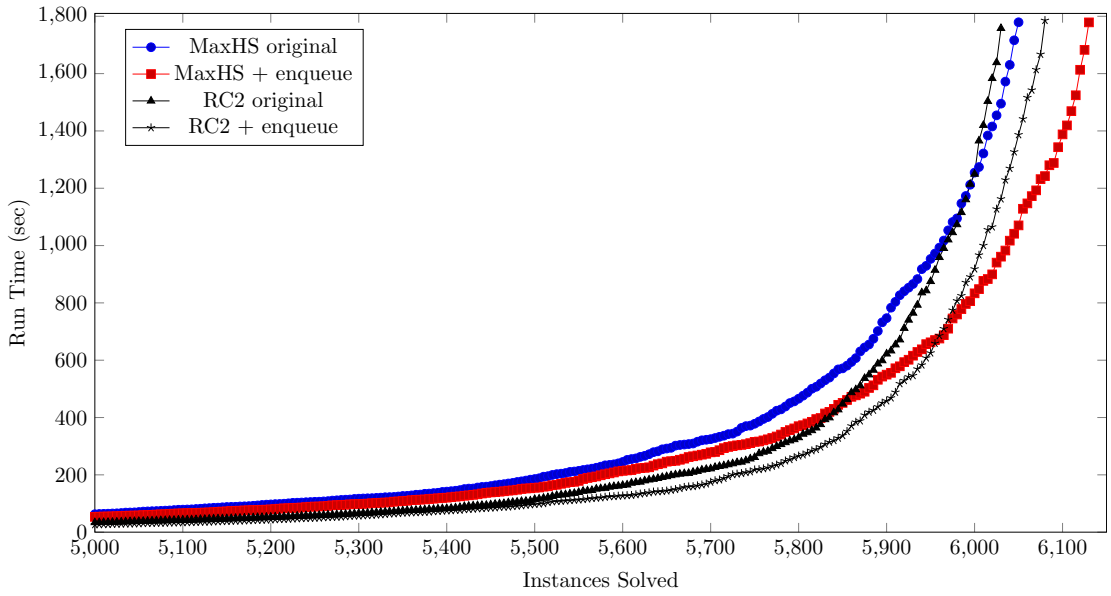

Figure 3.4: Cactus plot comparing total instances solved within a given time bound for MaxHS and RC2 with and without our assumption enqueueing techniques. The first 5000 instances were solved in less than 50 seconds each, so that part of the plot is truncated. For both solvers, the enqueueing assumptions technique solves more instances faster and more total instances overall.

savings across different SAT solver calls, is even less impactful. Hence, in terms of number of instances solved, assumptions-level trail savings does not seem to be either positively or negatively significant, and the remaining experiments use only the enqueueing technique.

Figure 3.4 shows a cactus plot of the two solvers with and without enqueueing. The plot shows that enqueueing provides a general speedup for both solvers with more instances generally being solved at every time bound in the plot. Note the first 5000 instances were solved by both solvers in $\leq$ 50 seconds per instance, so we truncated that part of the plot to show more detail on the harder instances.

3.7.3 Measuring Speedup/Slowdown of Each SAT Call

The run times shown in Figure 3.4 are affected both by the speed of the SAT solver and by the sequence of conflicts returned by the different SAT solvers. That is, the SAT calls the MaxSAT solver performs diverges as the instance is solved. Hence, to get a more precise picture of the SAT solver speedup and the quality of conflicts obtained by our technique, we changed RC2 so that it always invokes both the original Glucose solver and then the modified Glucose solver with enqueueing. However, RC2 always uses the conflict returned by the original Glucose solver in its further processing. In this way, each version of the SAT solver is solving an identical sequence of SAT calls during the processing of each MaxSAT instance. We ran this modified version of RC2 on the same suite of 7439 MaxSAT instances, giving it 3600 seconds per instance to account for the doubled up SAT solving. From this setup we obtained a sequence of matched pairs of SAT solver calls where the standard and enqueueing versions of the SAT solver are both invoked to solve the same formula subject to the same set of assumptions and with the same history of previous calls.

We measured the CPU time each SAT solver call took in each matched pair, discarding those pairs where both solvers took less than 0.1 seconds. In particular, we did not compare the CPU times of SAT solver calls that were so fast that they were likely to be too noisy. This yielded 22,810 matched pairs of SAT solver calls.

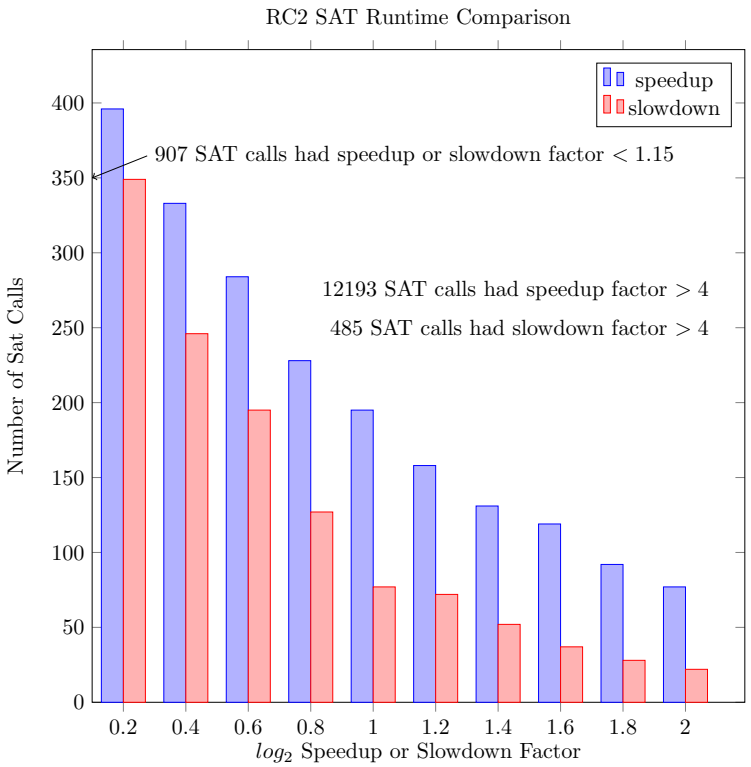

Figure 3.5: Comparison of number of SAT calls that had corresponding $\log_2$ speedup or slowdown factor for RC2 when assumption enqueueing technique was added. We grouped the speedup/slowdown factors into intervals, such that, e.g., the bar representing the number of instances that had a speedup factor of 0.2 is beside the number of instances that had a slowdown factor of 0.2. More instances are speeding up than slowing down in every interval.

There is quite a bit of variance in the runtimes, so a scatter plot of these points was not very informative. Instead for each pair we computed $\log_2(\frac{\text{Original Glucose CPU}}{\text{Enqueueing Glucose CPU}})$. This number is positive when original Glucose is slower and symmetric but negative when original Glucose is faster. Hence, the absolute value of the negative numbers represent instances that had that $\log_2$ speedup ratio, while the positive numbers represent instances that had that $\log_2$ slowdown ratio. Figure 3.5 shows a side-by-side histogram of the number of instances that had similar amounts of $\log_2$ speedup and slowdown ratio. The figure shows that although there is considerable variance among the instances—some were sped up while others were slowed down—there is a general trend that for all bands of speedup/slowdown factors more instances had a speedup of that factor than had a slowdown of that factor.

3.7.4 Measuring Growth/Shrinkage of Each Core

Our setup from Section 3.7.3 also yielded 949,003 matched pairs of conflicts for each solver. In each matched pair one conflict was produced by original Glucose and the other one by enqueueing Glucose, both from solving the identical SAT problem under an identical history of previous calls. As with the run times, we show in Figure 3.6 a side-by-side histogram of the $\log_2(\frac{\text{Size of conflict from Original Glucose}}{\text{Size of conflict from enqueueing Glucose}})$ growth and shrinkage ratios: shrinkage indicates that enqueueing Glucose produces a shorter conflict, while growth indicates that it produces a longer conflict. As with the run times there is a considerable variance in core sizes among

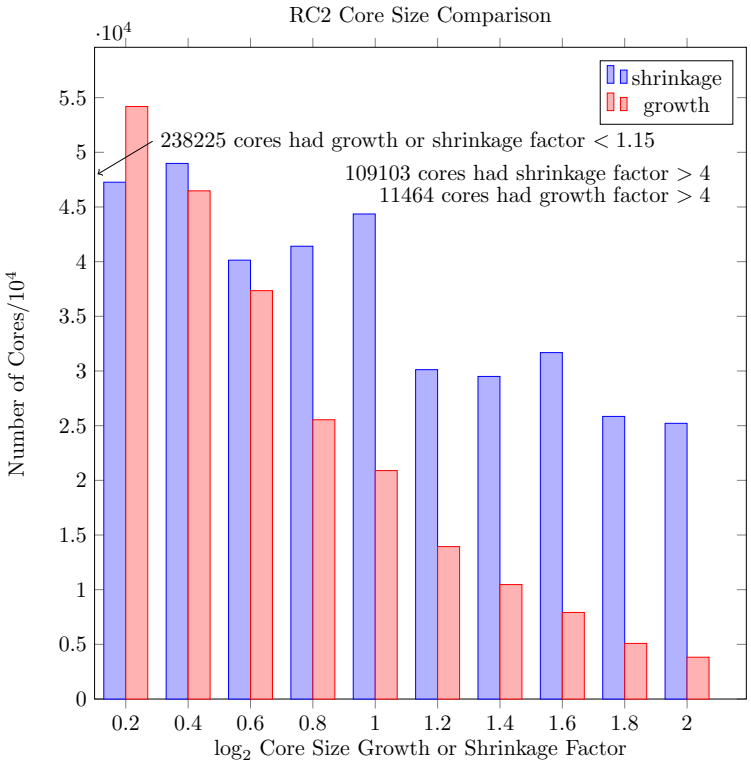

Figure 3.6: Individual core size $\log_2$ growth or shrinkage factor comparison for matching pairs of cores when running RC2. By matching pairs of cores, we are referring to the core that the solver with standard assumptions produces matched with the core that the solver with our assumptions technique produces. We grouped the growth/shrinkage factors into intervals, such that, e.g., the bar representing the number of cores that had a growth factor of 0.2 is beside the number of cores that had a shrinkage factor of 0.2. More cores shrank than grew for every factor interval except the first (i.e., 0.2).

these identical SAT calls—on some calls enqueueing Glucose produced larger conflicts, and on others it produced smaller conflicts. Nevertheless, the general trend is that for any band of growth/shrinkage ratio, more conflicts produced by enqueueing Glucose had that amount of shrinkage than that amount of growth. The only divergence from this trend was that there were more cores grown by a $\log_2$ factor between 0.1 to 0.3 (the band centered at 0.2) than shrunk by this factor. Most notably, however, almost 10 times as many cores were more than a factor of 4 smaller than were more than a factor of 4 larger.

This experiment shows that the overall better performance of enqueueing Glucose in RC2 is likely a product of both faster SAT calls and smaller conflicts. Together these two effects tend to make RC2 more effective. Although we do not have similar data for minisat used in MaxHS, we expect that similar results hold since MaxHS is also more effective with enqueueing.

3.7.5 MUSer Experiments

To illustrate that the enqueueing technique works not only for MaxSAT, but for another application of incremental SAT as well, we implemented our technique into MUSer [16], a well-known solver for extracting minimum unsatisfiable sets (MUS), by replacing its original SAT oracle, minisat, with minisat-enqueue. We will refer to the version of MUSer using minisat-enqueue for its SAT oracle as MUSer-enqueue. We ran all MUS experiments on a machine with 2 CPUs and 256GB RAM shared among sixteen 2.7GHz Intel Haswell cores. We ran MUSer and MUSer-enqueue, with a time limit of 30 minutes and a memory limit of 10GB, on the 300 benchmarks from the MUSer track at the 2011 SAT competition (the last time the MUS track was held at the SAT competition).

Both MUSer and MUSer-enqueue solved the same 271 instances. Figure 3.8 shows the average run times and Figure 3.7 compares the run times of each instance using a log ratio histograms generated in the same way as previous log ratio histograms in Section 3.7.3. The figures show that, although each solver is solving the same number of instances, MUSer-enqueue is usually solving the instances slightly faster.

In the benchmark suite we used for Muser, we were not able, however, to solve any additional instances. The heavier-weight techniques in the paper by Lagniez et al. [64] (involving introducing new abbreviation literals) were able solve additional instances mainly by reducing Muser's memory footprint. Similarly, although the data presented above about number of instances solved does not convincingly demonstrate the effectiveness of our proposed assumptions-level trail saving techniques, the finer grained data in 3.7 does indicate that these ideas do tend to yield run time speedups.

3.8 Discussion

In this chapter, we identified some redundant work that was occurring within the unit propagation process of assumptions in assumption-based SAT solving, and introduced some simple ideas for making the unit propagation of assumptions more efficient. We gave an example of how the simplest of our techniques, enqueueing all assumptions at once in Section 3.3, can result in a quadratic speedup of the unit propagation of assumptions. Our experiments then showed that enqueueing all assumptions at once is quite effective in improving MaxSAT solvers and shows some modest speedups for MUS solvers. Further experiments doing a detailed comparison of run times between our technique and the standard approach confirm that the improvements realized in the experiments was in fact a result of speeding up the unit propagation process, as opposed to something unintended. Similar detailed analysis comparing core sizes produced by our techniques

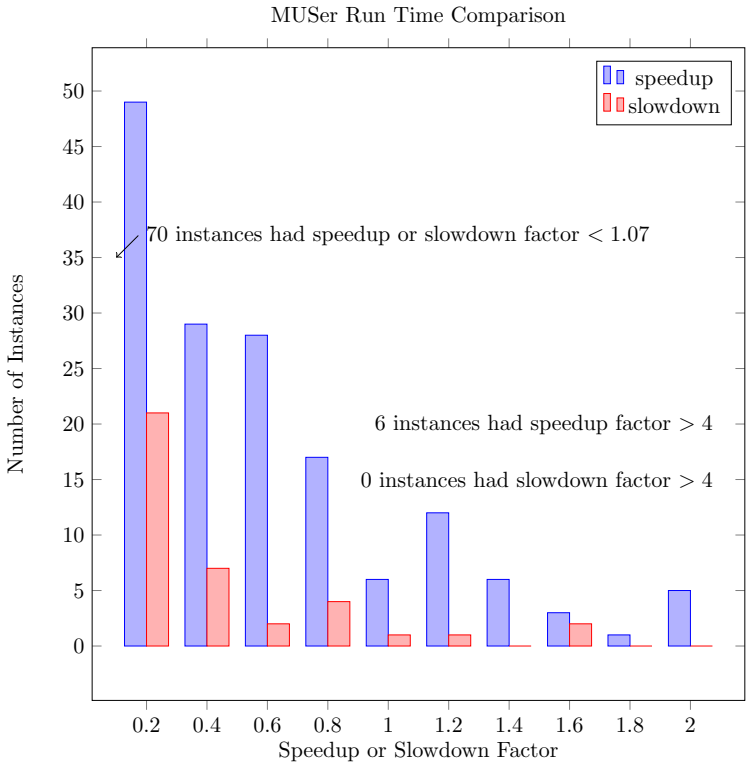

Figure 3.7: Log ratio comparison of speedup or slowdown factor for MUSer when using our assumptions enqueueing technique. We grouped the speedup/slowdown factors into intervals, such that, e.g., the bar representing the number of instances that had a speedup factor of 0.2 is beside the number of instances that had a slowdown factor of 0.2. More instances are speeding up than slowing down for every interval.

	instances solved	average time (s)
muser	271	141.0
muser-enqueue	271	124.3

Figure 3.8: Number of instances solved and average solve time for muser and muser-enqueue (i.e., muser with our assumptions enqueueing technique added). Our enqueueing assumptions technique did not help muser solve more instances, but led to a faster average solve time by more than 10%.

and the standard approach show something very surprising, that our technique produces smaller cores on average. This can have an exponential impact on the solver, and is likely also a contributor to the performance improvements realized in the experiments.

We also introduced some new ideas to reuse the trail for the assumption levels, which we called assumptions-based trail savings. These ideas only resulted in a modest improvement in the performance of the MaxSAT solver.

3.8.1 Future Work

Although the amount of improvement gained from the assumptions-level trail saving in Section 3.4 ideas was not as significant as our main technique, it is worth noting that we implemented the ideas in their most generic form without adding any parameters or conditions. There are a number of parameters one can introduce such as only applying some or all of the techniques when the total number of assumptions is greater than some threshold, or to cease trying to restore literals once a certain amount of the previous trail has failed to be restored. A future direction would be to introduce these parameters and experimentally determine the best setting to maximize the benefit of these ideas.

Another future direction could be to apply the ideas from this chapter in other contexts which use assumption-based SAT solving. Follow-up work that will be presented in the next chapter of this thesis will show how extending the assumptions-level trail savings ideas from this chapter so that they can work at all decision levels, instead of just at the assumption decision levels, can improve state-of-the-art SAT solvers. A similar extension of the assumption enqueueing ideas or assumptions-level trail saving ideas to propositional logic solvers beyond SAT, such as reachability checkers [76] or pseudo-boolean solvers [77], would be interesting to try. Assumption-based truth maintenance systems [78] are another application where assumption-based SAT can be used, albeit slightly differently, and it would be interesting to test whether our techniques improve performance in that setting as well.

Finally, Example 5 indicates that we do not always find a shorter conflict when we use our technique of enqueueing all the assumptions at once. Finding a way of incorporating clause minimization [79] into our technique might yield even shorter conflicts, and thus probably improve the solver's performance even further. We will leave this, and other investigations into why and when shorter conflicts arise from different propagation schemes, to future work.

Chapter 4

Trail Saving on Backtrack

4.1 Introduction

4.1.1 Motivation

In a CDCL SAT solver, backtracking occurs after every conflict, where all literals from one or more decision levels become unassigned before the solver resumes making decisions and performing unit propagations. Traditionally, SAT solvers would backtrack to the conflict level, which is the second highest decision level remaining in the conflict clause after conflict analysis has resolved away all but one literal from the current decision level [80]. Recently, however, it has been shown that partial backtracking [81] or chronological backtracking (i.e., backtracking only to the previous level after conflict analysis) [82, 83] can be effective on many instances. Chronological backtracking has been used in the solvers that won the 2018 and 2019 SAT Competitions [84]. Although chronological backtracking breaks some of the conventional invariants of SAT solvers, it has been formalized and proven correct [83] (also see related formalizations [85, 86]).

The motivation for using chronological backtracking is the observation that when a solver backtracks across many levels, many of the literals that are unassigned during the backtrack might be re-assigned again in roughly the same order when the solver redescends. This observation was first made in the context of restarts by van der Tak et al. [72]. Their technique backtracks to the minimum change level, i.e., the first level at which the solver's trail can change on redescent. However, their technique cannot be used when backtracking from a conflict: the solver's trail is going to be changed at the backtrack level so the minimum change level is the same as the backtrack level.

Chronological backtracking or partial backtracking instead allows a reduction in the length of the backtrack by placing literals on the trail out of decision level order. By reducing the length of the backtrack the solver can keep more of its assignment trail intact. This can save it from the work involved in reconstructing a lot of its trail, which is redundant work and often repeated many times due to the tendency of SAT solvers to stay in a similar part of the search space for a while (especially with the use of phase saving heuristics).

Using chronological backtracking is not a panacea however. Its application must be limited for peak effectiveness. This indicates that it is sometimes beneficial for the solver to backtrack fully and redo its trail, even if this takes more work. We will expand on why this might be the case below in Section 4.3, but three of the main side effects chronological backtracking could have on the search are: 1) literals are made unit at earlier decision levels that chronological backtracking can not detect, 2) different decisions get made

with chronological backtracking due to the heuristics changing (e.g., VSIDS scores change), or 3) different reasons do not get a chance to get assigned to literals that stay on the trail during chronological backtracking. *One of the main contributions of this chapter is to introduce a technique that saves the solver from having to do the redundant work described, while also not causing these aforementioned side effects on the search.*

4.1.2 Summary of Contributions

In this chapter, we present a new trail saving method whereby we save the backtracked part of the solver's trail and attempt to use that information to make the solver's redescent more efficient. This method can be seen as an extension of the "trail savings" idea from Section 3.4, except instead of only working during assumptions decisions levels, our new method will work on all decision levels throughout any SAT solving run. Unlike chronological backtracking, our trail saving method preserves the traditional invariants of the SAT solver and its basic version is very simple to implement. It allows the search to retain complete control over the order of decisions, but helps make propagation faster. We demonstrate experimentally that trail saving performs as well as and often better than chronological backtracking.

We then develop some enhancements to make trail saving more effective, including saving and appending trails over multiple backtracks, optionally rejecting usage of the trail when "reason quality" is poor, and using the trail to do a quick "lookahead" for conflicts. We show experimentally that with our enhancements we are able to further improve the performance of state-of-the-art SAT solvers.

4.1.3 Chapter Outline

In the remainder of this chapter, we will first give some background and develop some notation that will make it easier to discuss properties of the solver's assignment trail. We will then discuss chronological backtracking and highlight some of its strengths and deficiencies, before going on to introduce our new technique in Section 4.4. After proving the soundness of trail saving, we will go on to introduce three enhancements in Section 4.4.4 to trail saving and, finally, provide some experimental results that demonstrate the effectiveness of our new techniques. We will also provide a brief application study, showing how trail saving seems to be particularly useful on planning instances.

4.2 Background

We assume the reader is familiar with conflict-driven clause learning (CDCL) SAT solvers [69] and the concepts of unit propagation and conflict analysis. Please refer to Chapter 2 of this thesis and Chapter 4 of the Handbook of Satisfiability [70] for more detailed explanations. Below we will introduce some more specific notation and definitions that will be useful for the remainder of this chapter.

4.2.1 Trails

CDCL SAT solvers maintain a trail which is the sequence of literals that have currently been assigned $true$ by the solver. During its operation a SAT solver will add newly assigned literals to the end of the trail, and on backtrack remove literals from the end of the trail. For convenience, we will regard *literals as having been assigned* $true$ *if and only if they are on the trail*. So removing/adding a literal to the trail is equivalent to unassigning/assigning the literal $true$.

A SAT solver's trail satisfies a number of conditions. However, in this chapter we will need some additional flexibility in our definitions, as we will sometimes be working with trails that would never be constructed by a SAT solver. Hence, we define a *trail* to be a sequence of literals each of which is either a *decision* literal or an *implied* literal, and each of which has a *reason*. These two types of literals are distinguished by their *reasons*. Decision literals d have a null reason, $reason(d) = \varnothing$. Implied literals l have as a reason a clause of the formula $\mathcal{F}$, $reason(l) = C \in \mathcal{F}$. (The clause $reason(l)$ can be a learnt clause that has been added to $\mathcal{F}$).

If literal ℓ is on the trail $\mathcal{T}$ let $\iota_{\mathcal{T}}(\ell)$ denote its index on the trail, i.e, $\mathcal{T}[\iota_{\mathcal{T}}(\ell)] = \ell$. If x and y are both on the trail and $\iota_{\mathcal{T}}(x) < \iota_{\mathcal{T}}(y)$ we say that *x appears before y on the trail*. For convenience, when the trail being discussed is clear from context we simply write ι instead of $\iota_{\mathcal{T}}$.

Each literal $\ell \in \mathcal{T}$ has a decision level $decLvl(\ell)$ which is equal to the number of decision literals appearing on the trail up to and including ℓ ; hence, $decLvl(d) = 1$ for the first decision literal $d \in \mathcal{T}$. The set of literals on $\mathcal{T}$ that have the same decision level forms a *contiguous* subsequence[1] that starts with a decision literal d_i and ends just before the next decision literal d_{i+1}. We will often need to refer to different decision level subsequences of $\mathcal{T}$. Hence, we let $\mathcal{T}[[i]]$ denote the subsequence of literals at decision level i; and let $\mathcal{T}[[i \dots j]]$ denote the subsequence of literals at decision levels k for $i \leq k \leq j$.

Definition 1 *A clause C has **been made unit by $\mathcal{T}$ implying** l when $l \in C \wedge \big(\forall x \in C \;\; x \neq l \rightarrow \neg x \in \mathcal{T}\big)$. That is, all literals in C except l must have been falsified by $\mathcal{T}$.*

Note that we allow for the possibility of l appearing on the trail multiple times and allow for the possibility of both l and $\neg l$ appearing on the trail in this definition. Even though this will not happen to the main trail during the execution of the solver, this can happen when concatenating the trail with another sequence of literals, so we leave room to accommodate that in our definition.

Now we define the following properties that a trail $\mathcal{T}$ can have.

non-contradictory: A variable cannot appear in both polarities in the trail: $l \in \mathcal{T} \rightarrow \neg l \notin \mathcal{T}$.

non-redundant: A literal can only appear once on $\mathcal{T}$.

reason-sound: For each implied literal $l \in \mathcal{T}$ we have that its reason clause $reason(l) = C$ has been made unit by $\mathcal{T}$ implying l, and for each $x \in C$ with $x \neq l$ we have that $\neg x$ appears before l on $\mathcal{T}$: $\forall l \in \mathcal{T},\ reason(l) \neq \varnothing \rightarrow l \in reason(l) \wedge \big(\forall x \in reason(l).\, x \neq l \rightarrow \neg x \in \mathcal{T} \wedge \iota(\neg x) < \iota(l)\big)$. Note that we allow for the possibility of l appearing on the trail multiple times and allow for the possibility of both l and $\neg l$ appearing on the trail in this definition. Even though this will not happen to the main trail during the execution of the solver, this can happen when concatenating the trail with another sequence of literals, so we leave room to accommodate that in our definition.

propagation-complete: Unit propagation has been run to completion at all decisions levels of $\mathcal{T}$. This means that literals appear on $\mathcal{T}$ at the first decision level they were unit implied. Formally, this can be captured by the condition: $\forall i \in \{decLvl(l) \mid l \in \mathcal{T}\}.\big(\exists C \in \mathcal{F}.C$ is made unit by $\mathcal{T}[[0 \dots i]]$ implying $l\big) \rightarrow l \in \mathcal{T}[[0 \dots i]]$. Note that propagation completeness implies that $reason(l) \neq \varnothing$ must contain at least one other literal $y \neq l$ with $decLvl(y) = decLvl(l)$.

conflict-free: No clause of F is falsified by $\mathcal{T}$. Clauses $C \in F$ falsified by $\mathcal{T}$ are typically called *conflicts*.

[1] Our approach uses standard trails in which the decision levels are contiguous. Chronological backtracking [81, 82, 83] generates trails with non-contiguous decision levels

In CDCL solvers using standard conflict directed backtracking, all properties hold of the prefix of the solver's trail consisting of all decisions levels but the deepest. The full trail might, however, contain a conflict at its deepest level, so is not necessarily conflict-free. The full trail might also not be propagation-complete, as unit propagation at the deepest level is typically terminated early if a conflict is found. It can further be noted that the first four properties imply that if a clause C is falsified at decision level k, then C must contain at least two literals at level k (otherwise C would have become unit at a prior level and then satisfied by making its last unfalsified literal $true$).

4.2.2 Standard Backtracking

In CDCL SAT solving the solver extends its trail by adding new decision literals followed by finding and adding all unit implied literals arising from that new decision. This continues until it reaches a decision level L_{deep} where a conflict C is found.

In standard backtracking, the solver then constructs a new 1-UIP clause by resolving away all but one literal at level L_{deep} from the conflict C using the reason clauses of these literals. (As noted above C must contain at least two literals at level L_{deep}). Hence, the new clause $C_{1\text{-}UIP}$ will contain one literal ℓ_{deep} at level L_{deep} and have all of its other literals a levels less than L_{deep}. The solver then backtracks to L_{back} the second deepest level in $C_{1\text{-}UIP}$. This involves changing $\mathcal{T}$ to its prefix $\mathcal{T}[[0 \dots L_{back}]]$ (by our convention all literals removed from $\mathcal{T}$ are now unassigned). The new clause $C_{1\text{-}UIP}$ is made unit by $\mathcal{T}[[0 \dots L_{back}]]$ implying ℓ_{deep}, so the solver then adds ℓ_{deep} to the trail and executes another round of unit propagation at level L_{back}, after which it continues by once again growing the trail with new decisions and unit implied literals until a new conflict or a satisfying assignment is found.

In standard backtracking, the difference between the backtrack level, L_{back} and the current deepest level L_{deep} can be very large. During its new descent from L_{back} the solver can reproduce a large number of the same decisions and unit propagations, essentially wasting work. This potential inefficiency has been noted in prior work [72, 81, 82, 83].

In a paper by van der Tak et al. [72] a technique for reducing the length of the backtrack during restarts was presented. In restarts, the solver backtracks to level 0, and this technique involves computing a new deeper backtrack level $M > 0$ for which it is known that on redescent the first $M + 1$ levels of the trail will be unchanged (except perhaps for the ordering of the literals). This technique removes the redundant work of reproducing the first M trail levels. When backtracking from a conflict, however, the trail will be changed at level L_{back} (ℓ_{deep} will be newly inserted at this level). Hence this technique cannot reduce the length of the backtrack. In this chapter we will show that although we have to backtrack to L_{back} we can make the subsequent redescent much more efficient.

4.2.3 Chronological Backtracking

Chronological backtracking and partial backtracking in the context of clause learning solvers are alternatives to standard backtracking which allow the solver to execute a shorter backtrack. That is, with these techniques the solver can avoid having to go all the way back to the second deepest level in the learnt clause, as in standard backtracking.

Formalisms for partial backtracking in clause learning solvers have been presented in previous papers [85, 86]. In a different paper by Jiang et al. [81], practical issues of implementation were addressed, and experiments with a CDCL solver using partial backtracking were shown. In a paper by Nadel et al. [82], improved

and more efficient implementation techniques were developed which allowed chronological backtracking to make improvements to state-of-the-art SAT solvers. Finally, a paper by Möhle et al. [83] presented additional implementation ideas and details along with correctness results for these methods.

There has been previous work on finding a smart point to backtrack to during a backtracking search in other contexts [87, 88]. Chronological backtracking and partial backtracking in the context of SAT solving are highly related to dynamic backtracking [89] and partial order backtracking [90] in the context of constraint satisfaction solving.

The aim of partial backtracking is to reduce the redundant work that might be done by the SAT solver on its redescent from the backtrack level L_{back}. The technique allows the solver to backtrack to any level j in the range $L_{back} \leq j \leq L_{deep}-1$ (where L_{deep} is the level the conflict was discovered). Nadel and Ryvchin [82] proposed to always backtrack chronologically to $L_{deep}-1$ while Möhle and Biere [83] returned to the proposal by Jiang et al. [81] of flexibly backtracking to any level in the allowed range. Note that the new learnt 1-UIP clause $C_{1\text{-}UIP}$ is still unit at every level in this range. So after backtracking to level j the newly implied literal ℓ_{deep} is added to the trail with $reason(\ell_{deep}) = C_{1\text{-}UIP}$, and $decLvl(\ell_{deep})$ is set to L_{back} (the second deepest level in $C_{1\text{-}UIP}$).

This means that the decision levels on the trail are no longer contiguous, as ℓ_{deep} has a different level than the other literals at level j (if $j \neq L_{back}$). This change has a number of consequences for the SAT solver's operation, all of which were described in the paper by Jiang et al [81]. Möhle and Biere [83] showed that despite these consequences partial backtracking can be made to preserve the soundness of a CDCL solver.

4.3 Chronological Backtracking Effects on Search

We are presenting a new technique that allows the SAT solver to use standard backtracking, but also allows saving some redundant work on its redescent. Our method has more overhead than chronological backtracking, so the first question that must be addressed is, why not just use chronological backtracking? We will address that question in this section.

4.3.1 Chronological Backtracking Most Effective With Limited Application

Although chronological backtracking is able to avoid a lot of redundant work it also has other effects on the SAT solver search. These effects are sometimes detrimental to the solver's performance and so it is not always beneficial to use chronological backtracking. In fact, in both recent papers on the subject [82, 83], it was found that fairly limited application of chronological backtracking performed best. In the paper by Nadel and Ryvchin [82], chronological backtracking was applied only when the length of the standard backtrack, $L_{back} - L_{deep}$ was greater than a given threshold T. In their experiments they found that $T = 100$ was the best value, i.e., chronological backtracking is done only on longer backtracks. In practice, this meant that chronological backtracking was relatively infrequent; in our measurements with their solver only about 3% of the solver backtracks were chronological backtracking backtracks. In the paper by Mohle and Biere [83], the value $T = 100$ was also applied. However, they introduced an additional technique to add some applications of chronological backtracking when the length of the backtrack is less than T. This allowed them to utilize chronological backtracking in about 15% of the backtracks.

4.3.2 Potential Side Effects of Chronological Backtracking

Although it is difficult to know precisely why chronological backtracking is not always beneficial, we can identify some different ways in which chronological backtracking can affect the SAT solver's search. With standard backtracking the literal ℓ_{deep} is placed on the trail at the end of L_{back} and then unit propagated. This could impact the trail in at least the following ways:

1. First, some literals might become unit at earlier levels. This could include decision literals becoming forced which might merge some decision levels.

2. Second, different decisions might be made due to changes in the variable scores arising from the newly learnt clause.

3. And third, literals might be unit implied with different reasons.

Chronological backtracking can change all of these things, each of which could have an impact on the future learnt clauses, and thus on the solver's overall efficiency.

The second impact, changing variable scores, is partially addressed in the paper by Mohle and Biere [83] who utilize the ideas of van der Tak et al. [72] to backtrack to a level where the decisions would be unchanged. However, if the length of the backtrack is greater than T there could still be a divergence between the variable decisions generated in standard backtracking and chronological backtracking.

An argument is also given in a paper by van der Tak et al. [72] that the third impact, changing literal reasons, is not significant. However, the experiments in that paper were run before good notions of clause quality were known [91]. Our empirical results indicate that once clause quality is accounted for, changing the literal reasons can have a significant impact.

The first impact is worth discussing since it was mentioned in the paper by Jiang et al. [81] but not in the subsequent works. This is the issue of changing the decision levels of literals on the trail. Chronological backtracking computes the decision level of each implied literal based on the decision levels of the literals in its reason, but it does not go backwards to change the decisions levels of literals earlier on the trail. Example 9 below illustrates this issue.

Example 9 *Suppose that $(x, \neg y) \in \mathcal{F}$, the literal x is a decision literal on the trail with $decLvl(x) = 2$, and that the solver is currently at level 150 where it encounters a conflict. If this conflict yields the unit clause (y), standard backtracking would backtrack to level 0, where x would be implied. On redescent, x would no longer form a new decision level and it would not appear in any new clauses (as it is entailed by $\mathcal{F}$). Chronological backtracking, on the other hand, would backtrack to level 149. On its trail x would still be at level 2. Until a backtrack past level 2 occurs, learnt clauses might contain $\neg x$, and thus have level 2 added to their set of levels (potentially changing their LBD score). Only when a backtrack past level 2 occurs would x be restored to its correct level 0, and it would require inprocessing simplifications [92] to remove x from the learnt clauses.* ■

In sum, although these impacts of chronological backtracking on the SAT solver's search might or might not be harmful to the SAT solver, they do exist. In fact, there are two pieces of evidence that these impacts can sometimes be harmful. First, as mentioned above, previous work found that it is best to only apply chronological backtracking on large backtracks where it has the potential to save the most work. If there were no harmful effects it would always be effective to apply chronological backtracking. And second, in our empirical results below we show that our new trail saving technique, which always uses standard

backtracking, can often outperform chronological backtracking. Although our technique reduces the solver's work on redescent it does not completely eliminate it like chronological backtracking does. Hence its superior performance can only occur if chronological backtracking is sometimes harmful.

4.3.3 Combination of Chronological Backtracking and Trail Saving

It is possible to combine chronological backtracking with our trail saving technique to reduce the amount of work required whenever the solver performs non-chronological backtracking. However, chronological backtracking greatly reduces the potential savings that could be achieved by our method since most of its non-chronological backtracks are relatively short (less than threshold T levels). In our preliminary experiments this combination did not seem promising.

Nevertheless, there is good evidence that chronological backtracking can improve SAT solver performance.[2] and it seems to be the case that it is better to perform chronological backtracking in some branches of the search tree. Hence, an interesting direction for future work would be to develop better heuristics about when to use chronological backtracking in a branch and when to use standard backtracking augmented by our trail saving method.

4.4 Trail Saving

In this section, we will outline our new approach, *trail saving*, precisely, prove its correctness, and show how to implement it in a modern SAT solver. Some examples are provided to illustrate how trail saving works and why it can save time during unit propagation. We will then describe some enhancements in Section 4.4.4 to trail saving that we developed to try to further improve its performance.

4.4.1 Approach

Our approach is to save the trail $\mathcal{T}$ on backtrack, and to use the saved trail $\mathcal{T}_{save}$ when the solver redescends to improve the efficiency of propagations without affecting the decisions the solver wants to make. The saved trail $\mathcal{T}_{save}$ also provides a secondary "lookahead mechanism" that the SAT solver can exploit as it redescends.

Suppose that the solver is at L_{deep} where it has encountered a conflict. From the 1-UIP clause it learns, $C_{1\text{-}UIP}$, it now has to backtrack to L_{back}. This is accomplished by calling BACKTRACK(L_{back}), shown in Figure 4.1, which saves the backtracked portion of the trail.

Note that BACKTRACK does not save the deepest level of $\mathcal{T}$. The full $\mathcal{T}$ contains a conflict (at its deepest level). Hence the solver will never reproduce all the same levels, and it would be useless to save all of them. Note also that in addition to saving the literals in $\mathcal{T}_{save}$, we also save the clause reason of the unit implied literals in a separate $reason_{save}$ vector. Finally, we see that after backtrack the first literal on $\mathcal{T}_{save}$ is a decision literal: it is the first literal of $\mathcal{T}$ at decision level $L_{back} + 1$. Literals will be removed from $\mathcal{T}_{save}$ during its use, but always in units of complete decision levels. So $\mathcal{T}_{save}[0]$ will always be a (previous) decision literal.

After backtrack the solver will add ℓ_{deep} to the end of the updated $\mathcal{T}$ with $reason(\ell_{deep}) = C_{1\text{-}UIP}$ and then invoke unit propagation. $\mathcal{T}_{save}$ is exploited during propagation by the version of PROPAGATE shown

[2]Chronological backtracking can also be extremely useful in contexts where each descent can be very expensive, e.g., when doing theory propagation in SMT solving [93], or component analysis in #SAT solving [6]. In these cases, chronological backtracking, by avoiding backtracking and subsequent redescent, has considerable potential for improving solver performance.

in Figure 4.1, which will initially be invoked with the argument $\iota(\ell_{deep})$ (i.e., the trail index of the newly added implicant). The saved trail will be continually consulted during the solver's descent whenever unit propagation is performed. When backtrack occurs, $\mathcal{T}_{save}$ will be overwritten to store the new backtracked portion of $\mathcal{T}$.

$\mathcal{T}_{save}$ is consulted in the procedure USESAVEDTRAIL (Figure 4.1). This procedure tries to add saved implied literals and their reasons to the solver's trail, when these implications are valid. We will show below that those implications that are added are in fact valid. We do not interfere with the solver's variable decisions. Instead we opportunistically test to see if literals implied on $\mathcal{T}_{save}$ are valid implications for the solver given the solver's current decisions.

$\mathcal{T}_{save}[0]$ is always a (previous) decision literal d with $reason_{save}(d) = \varnothing$. Note that, since new literals (e.g., ℓ_{deep}) have been added to $\mathcal{T}$, d might now be an implied literal on $\mathcal{T}$ (i.e., $reason(d) \neq \varnothing$) even though before the backtrack it was previously a decision (i.e., $reason_{save}(d) = \varnothing$). If d has been assigned $false$ by the solver (i.e., $\neg d \in \mathcal{T}$), we cannot add any implied literals below it on $\mathcal{T}_{save}$ to $\mathcal{T}$ as these implied literals depend on d being assigned $true$. In this case we stop looking for more literals to add to $\mathcal{T}$ (line 25).

On the other hand if d has been made $true$ by the solver we can continue to add all of the implied literals below it (up to but not including the next decision literal on $\mathcal{T}_{save}$) to $\mathcal{T}$ (line 33), reusing their saved reasons. Any literals that have already been made true by the solver can be skipped (line 28). Finally, if we encounter a literal that has already been falsified by the solver, then its saved reason clause must be falsified by the solver and we can return it as a conflict (line 30). If a conflict is encountered, we leave $\mathcal{T}_{save}$ unchanged by resetting idx to zero. Otherwise, idx will be the number of literals at the front of $\mathcal{T}_{save}$ that have been moved to $\mathcal{T}$ (or skipped over since they are already on $\mathcal{T}$). We then remove the first idx literals from $\mathcal{T}_{save}$ (line 36), and return the conflict (equal to $\varnothing$ if no conflict was found).

An illustration of this entire trail saving process is given below in Example 10.

Example 10 *Figure 4.2 provides an example of how $\mathcal{T}_{save}$ is used. Initially the literals l_1 to l_{14} are on the solver's $\mathcal{T}$, and $\mathcal{T}_{save}$ is empty. This is shown in the first two lines of the figure. In the figure the superscript on the literals indicates their decision level, and a superscripted $*$ indicates that the literal is a decision. Hence l_1^{1*} indicates that $decLvl(l_1) = 1$ and that l_1 is a decision.*

Then a conflict is found at level 6 and the 1-UIP clause $(\neg l_1, \neg l_3, \neg l_{12})$ is learnt. Thus the solver will backtrack to level 2, where it will add $\neg l_{12}$ as a unit implicant. The next two lines show $\mathcal{T}$ and $\mathcal{T}_{save}$ right after the backtrack to level 2: the backtracked levels have been copied into $\mathcal{T}_{save}$ omitting the conflict level 6.

The new unit $\neg l_{12}$ is now added to $\mathcal{T}$ and unit propagation performed adding l_7 and l_9 to level 2. Since the first literal on $\mathcal{T}_{save}$, l_5, has $reason_{save}(l_5) = \varnothing$ (l_5 was a decision on $\mathcal{T}$ at the time backtrack occurred) and is not yet $true$, $\mathcal{T}_{save}$ is not helpful at this stage. The status of $\mathcal{T}$ and $\mathcal{T}_{save}$ at this point is shown in the figure.

After unit propagation is finished the solver makes a new decision, which happens to be (but is not forced to be) l_5. Now $\mathcal{T}_{save}$ can be used: l_5 is $true$ so it is removed, l_6 is unassigned so it is added to $\mathcal{T}$, l_7 is $true$ and so removed, l_8 is unassigned so it is added to $\mathcal{T}$, l_9 is $true$ and removed, and finally l_{10} and l_{11} are unassigned and so are added to $\mathcal{T}$. In this example, $\mathcal{T}_{save}$ is emptied, and cannot contribute more to $\mathcal{T}$.

All of these units are added to $\mathcal{T}$ before the solver starts to unit propagate l_5. Since, new literals have been added to $\mathcal{T}$ before l_5 the solver must propagate l_5 and all of the literals that follow it before making its next decision. ■

```
1: BACKTRACK(L_back)
2:     ∀ℓ ∈ 𝒯[[L_back+1 ... L_deep−1]] reason_save(ℓ) = reason(ℓ)
3:     𝒯_save = 𝒯[[L_back+1 ... L_deep−1]]
4:     𝒯 = 𝒯[[0 ... L_back]]
```

```
propagate ()
while idx < 𝒯.size() do
    c ← USESAVEDTRAIL ()
    if (c ≠ ∅) then
        return c                                   /* Found conflict from 𝒯_save */
    ℓ ← 𝒯[idx]
    for each clause c ∈ watchlist(¬ℓ) do
        if (c is unit implying x) then
            𝒯.addToEnd(x);
            reason(x) ← c
        else if (c is falsified) then
            return c                               /* Found conflict from unit prop. */
        else
            Update c's watches.
    idx++

useSavedTrail ()
idx ← 0; c ← ∅
for ( ; idx < 𝒯_save.size(); idx++) do
    l_save ← 𝒯_save[idx]
    if (reason_save(l_save) = ∅) then
        if (l_save ∈ 𝒯) then
            continue                               /* true in solver, we can use its implied lits */
        else
            break                                  /* Only solver can set decisions, so we stop here */
    else
        if (l_save ∈ 𝒯) then
            continue
        else if (¬l_save ∈ 𝒯) then
            /* contradiction, return conflict and reset idx */
            c ← reason_save(l_save); idx ← 0; break
        else
            𝒯.addToEnd(l_save)
            reason(l_save) ← reason_save(l_save)
for (i ← 0; i < idx; i++) do
    𝒯_save.removeFront()
return c
```

Figure 4.1: Using $\mathcal{T}_{save}$ in unit propagation and conflict detection

$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2	l_5^{3*}	l_6^3	l_7^{4*}	l_8^4	l_9^{5*}	l_{10}^5	l_{11}^5	l_{12}^{6*}	l_{13}^6	l_{14}^6
$\mathcal{T}_{save}$														
1-UIP ($\neg l_1$, $\neg l_3$, $\neg l_{12}$) and backtrack 2 (L_{back})														
$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2										
$\mathcal{T}_{save}$					l_5^*	l_6	l_7^*	l_8	l_9^*	l_{10}	l_{11}			
Unit Prop $\neg l_{12}$ (ℓ_{deep}), $\mathcal{T}_{save}$ not yet helpful														
$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2	$\neg l_{12}^2$	l_7^2	l_9^2							
$\mathcal{T}_{save}$					l_5^*	l_6	l_7^*	l_8	l_9^*	l_{10}	l_{11}			
Make decision														
$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2	$\neg l_{12}^2$	l_7^2	l_9^2	l_5^{3*}						
$\mathcal{T}_{save}$					l_5^*	l_6	l_7^*	l_8	l_9^*	l_{10}	l_{11}			
Now $\mathcal{T}_{save}$ can be used to augment trail														
$\mathcal{T}$	l_1^{1*}	l_2^1	l_3^{2*}	l_4^2	$\neg l_{12}^2$	l_7^2	l_9^2	l_5^{3*}	l_6^3	l_8^3	l_{10}^3	l_{11}^3		
$\mathcal{T}_{save}$														

Figure 4.2: Use of $\mathcal{T}_{save}$ from Example 10. The literal's decision level is indicated in its superscript, and a * superscript indicates that the literal is a decision. The highlighted text in between the trails indicates what the SAT solver is doing next.

4.4.2 Potential Benefits of Trail Saving

As noted in the previous example, unit propagation has to be rerun on all saved literals added to $\mathcal{T}$ from $\mathcal{T}_{save}$. Thus our technique, unlike chronological backtracking, does not completely remove the overhead of reproducing the trail on the solver's redescent. Nevertheless, trail saving improves the efficiency of this redescent in three different ways. First, by adding more forced literals to the trail before continuing propagating the next literal, propagation can potentially gain a quadratic speedup [94, 71]. Second, propagation does not need to examine the reason clause of the added literals. If these literals were not added by USESAVEDTRAIL, propagation would have to traverse each of these reason clauses to determine that they have in fact become unit. Third, when a conflict is returned by USESAVEDTRAIL all further propagations can be halted. The added literals and their reasons will be sufficient to perform clause learning from the conflict returned by USESAVEDTRAIL. Since trail saving can sometimes save hundreds or thousands of literals at a time, these savings can in sum be significant.

For completeness, we provide Example 11 below that illustrates how trail saving can lead to a speedup in unit propagation.

Example 11 *Suppose the literals $x_1, ...x_n$ are on $\mathcal{T}_{save}$, with none of $x_2, ...x_n$ having a null reason, and x_1 becomes true (e.g., the solver makes a decision for x_1 to be true). Then, $x_2, ...x_n$ are implied by the current trail. Now we will examine the propagation of a clause c which consists of the literals $l, \neg x_1, \neg x_2, ..., \neg x_n$ with and without trail saving. Assume l and $\neg x_1$ are the current watcher literals.*

Without trail saving, $\neg x_1$ becomes false and a new watcher is searched for and found 1 literal later, $\neg x_2$, and then $\neg x_1$ and $\neg x_2$ are swapped. Next $\neg x_2$ becomes false, a new watcher is searched for and found 2 literals later, $\neg x_3$, and then $\neg x_2$ and $\neg x_3$ are swapped. This process will continue until all x_n literals are false, c becomes unit, and l becomes forced to be true. This propagation took $1 + 2 + ... + n$ searches through clause c, for a total of $O(n^2)$ searches.

With trail saving, $x_2, ...x_n$ are all assigned to be true as soon as x_1 is made true, so immediately the values of literals $\neg x_1, ..., \neg x_n$ in c are false. During propagation, a new watcher is searched for to replace $\neg x_2$, n literals are searched over without finding any, thus c is detected as unit and l becomes forced to be true. This propagation took $O(n)$ searches through clause c. ■

4.4.3 Correctness

Now we will prove that our use of $\mathcal{T}_{save}$ preserves the SAT solver's soundness. In particular, $\mathcal{T}_{save}$ is only used in the procedure USESAVEDTRAIL, in which it either adds new literals to the solver's trail, or returns conflict clauses to the solver. Hence, we only need to show that these new literals are in fact unit implied and the conflicts are in fact falsified by the solver's trail. Since both $\mathcal{T}$ and $\mathcal{T}_{save}$ are sequences of literals (with associated reasons) we can consider their concatenation denoted as $\mathcal{T} + \mathcal{T}_{save}$.

Theorem 1 *For any $\mathcal{T}$ and $\mathcal{T}_{save}$, if $\mathcal{T} + \mathcal{T}_{save}$ is **reason sound**(Section 4.2) then the following holds. If the first i literals on $\mathcal{T}_{save}$ are all in $\mathcal{T}$ ($\forall j, 0 \leq j < i \rightarrow \mathcal{T}_{save}[j] \in \mathcal{T}$) and $\mathcal{T}_{save}[i] = l$ is an implied literal with $reason_{save}(l) = C$, then C has been made unit by $\mathcal{T}$ implying l.*

Proof: Since $\mathcal{T}+\mathcal{T}_{save}$ is reason sound, every literal in C other than l appears negated before l in the sequence $\mathcal{T} + \mathcal{T}_{save}$. Thus for $x \in C$ we have $\neg x \in \mathcal{T}$ or $\neg x \in \mathcal{T}_{save}[0] \ldots \mathcal{T}_{save}[i-1]$. But in the later case we also have $\neg x \in \mathcal{T}$. □

This theorem shows that USESAVEDTRAIL's processing is sound whenever $\mathcal{T} + \mathcal{T}_{save}$ is reason sound. In this procedure, an implied literal from $\mathcal{T}_{save}$ is added to $\mathcal{T}$ (line 33) only when all prior literals on $\mathcal{T}_{save}$ are already on $\mathcal{T}$ (i.e., previously on $\mathcal{T}$ or already added to $\mathcal{T}$). Thus each new addition is sound given the soundness of the previous additions. If l is to be added, Theorem 1 shows that every other literal in $reason_{save}(l)$ has been falsified by $\mathcal{T}$. Hence, if l is to be added but l is also falsified by $\mathcal{T}$, then $reason_{save}(l)$ is a clause falsified by $\mathcal{T}$, thus it is a sound conflict for the solver, and that is exactly how USESAVEDTRAIL handles that case.

Now we only have to show that $\mathcal{T} + \mathcal{T}_{save}$ is always **reason sound** during the operation of the solver.

Let T.removeFront() be a routine which removes and returns the top literal from a trail T.

Proposition 1 *If $\mathcal{T} + \mathcal{T}_{save}$ is reason sound then $\mathcal{T}' + \mathcal{T}'_{save}$ is reason sound in all of the following cases.*

1. *$\mathcal{T}_{save}[0] \in \mathcal{T}$, $\mathcal{T}' = \mathcal{T}$, and $\mathcal{T}'_{save} = \mathcal{T}_{save}.\mathrm{removeFront}()$.*
2. *$\mathcal{T}' = \mathcal{T} + \mathcal{T}_{save}[0]$ and $\mathcal{T}'_{save} = \mathcal{T}_{save}.\mathrm{removeFront}()$.*
3. *$\mathcal{T}' = \mathcal{T} + \mathcal{T}_{new}$ and $\mathcal{T}'_{save} = \mathcal{T}_{save}$ and $\mathcal{T}'$ is reason sound, where $\mathcal{T}_{new}$ is the sequence of literals added to $\mathcal{T}$ by the solver since the last backtrack.*
4. *We also have that $\mathcal{T}$ is reason sound if $\mathcal{T}$ was generated by the solver.*

Proof: (1) $\mathcal{T}_{save}[0]$ already appears earlier in the $\mathcal{T}$ so it can be removed without affecting the soundness of any reason following it. (2) is obvious as the sequence is unchanged. (3) the reasons in $\mathcal{T} + \mathcal{T}_{new}$ are sound by assumption. Those in $\mathcal{T}_{save}$ remain sound as they depend only on the literals in $\mathcal{T}$ and prior literals on $\mathcal{T}_{save}$, both of which are unchanged. (4) is obvious from the operation of unit propagation in the solver. □

Theorem 2 *$\mathcal{T} + \mathcal{T}_{save}$ is always reason sound during the operation of the solver.*

Proof: $\mathcal{T}_{save}$ starts off being empty, so $\mathcal{T} + \mathcal{T}_{save} = \mathcal{T}$ is reason sound as it was generated by the solver (4). In procedure BACKTRACK $\mathcal{T} + \mathcal{T}_{save}$ is set to a trail that was previously generated by the solver (4). The

solver can add to $\mathcal{T}$ by decisions and propagations without using $\mathcal{T}_{save}$. In this case $\mathcal{T}' = \mathcal{T} + \mathcal{T}_{new}$, and $\mathcal{T}'$ is reason sound by (4), thus the new $\mathcal{T}' + \mathcal{T}_{save}$ is reason sound by (3). Finally, in procedure USESAVEDTRAIL either (a) literals at the front of $\mathcal{T}_{save}$ are discarded since they already appear on $\mathcal{T}$, or (b) literals are moved from $\mathcal{T}_{save}$ to $\mathcal{T}$. Under both of these changes $\mathcal{T} + \mathcal{T}_{save}$ remains reason sound by (1) and (2). □

4.4.4 Enhancements

We developed three enhancements of the base trail saving method described above. In this section we present these enhancements.

Saving the trail over multiple backtracks

It can often be the case that when the solver finds a conflict and backtracks to L_{back} it might immediately find another conflict at L_{back} causing a further backtrack. In the procedure BACKTRACK every backtrack causes $\mathcal{T}_{save}$ to be overwritten. Hence, in these cases most of the trail will not be saved—only the portion from the last backtrack. Our first extension addresses this potential issue and also provides more general trail saving in other contexts as well.

This extension is simply to add the latest backtrack to the front of $\mathcal{T}_{save}$ leaving all of the previous contents of $\mathcal{T}_{save}$ intact. Specifically, we replace line 3 of BACKTRACK by the new line:

3. $\quad \mathcal{T}_{save} = \mathcal{T}[[L_{back}{+}1 \ldots L_{deep}{-}1]] + \mathcal{T}_{save}$

It is not difficult to show that this change preserves soundness. Only Theorem 2 is potentially affected. However, we know that $\mathcal{T}_{save}$ is unchanged at the level at which a conflict occurs: either the conflict is detected without consulting $\mathcal{T}_{save}$ or if the conflict comes from $\mathcal{T}_{save}$ then USESAVEDTRAIL leaves $\mathcal{T}_{save}$ unchanged (line 30). Hence, at the level before the conflict occurred we have inductively that $\mathcal{T}[[0 \ldots L_{deep}{-}1]] + \mathcal{T}_{save}$ was reason sound, and hence so is $\mathcal{T}' + \mathcal{T}'_{save}$ with $\mathcal{T}' = \mathcal{T}[[0 \ldots L_{back}]]$ and $\mathcal{T}'_{save} = \mathcal{T}[[L_{back}{+}1 \ldots L_{deep}{-}1]] + \mathcal{T}_{save}$.

When adding to the front of $\mathcal{T}_{save}$ in this manner $\mathcal{T}_{save}$ can grow indefinitely. So we prune $\mathcal{T}_{save}$ when it gets too large by (a) removing $\mathcal{T}_{save}[i]$ if $\mathcal{T}_{save}[i] = \mathcal{T}_{save}[j]$ for some $j < i$ ($\mathcal{T}_{save}[i]$ is redundant), and (b) removing the suffix of $\mathcal{T}_{save}$ starting at $\mathcal{T}_{save}[i]$ when $\mathcal{T}_{save}[j] = \neg\mathcal{T}_{save}[i]$ for some $j < i$ ($\mathcal{T}_{save}[i]$ will never be useful as its negation, $\mathcal{T}_{save}[j]$, would have to be added to $\mathcal{T}$ first). In this way $\mathcal{T}_{save}$ need never become larger than the number of variables in $\mathcal{F}$.

Lookahead for Conflicts

In USESAVEDTRAIL we stop adding literals from $\mathcal{T}_{save}$ to $\mathcal{T}$ once we reach a decision literal d on $\mathcal{T}_{save}$ that is not yet on $\mathcal{T}$ (line 25 of USESAVEDTRAIL). This is done so that the solver has full control over variable decisions without interference from the trail saving mechanism (unlike the case with chronological backtracking). However, another option would be to force the solver to use d as its next decision literal, which would then allow us to further add all of d's implied literals on $\mathcal{T}_{save}$ onto $\mathcal{T}$. This can be done for the first k decisions on $\mathcal{T}_{save}$ for any k. But in general, we do not want to remove the solver's autonomy by forcing it to make potentially different decisions than it might have wanted to.

However, if there is a literal $l \in \mathcal{T}_{save}$ for which $\neg l \in \mathcal{T}$, we can observe that forcing the solver to make all of the decisions of $\mathcal{T}_{save}$ that lie above l will immediately generate a conflict in the solver: $reason_{save}(l)$ will be falsified. In fact, in this situation we would not even need to perform unit propagation over the literals

added from $\mathcal{T}_{save}$; the literals and their reasons obtained from $\mathcal{T}_{save}$ would be sufficient to perform 1-UIP learning from $reason_{save}(l)$.

We experimented with this "lookahead for conflicts" idea using various values of k. We found that $k = 2$, i.e., forcing up to two decisions from $\mathcal{T}_{save}$ to be made by the solver, often enhanced the solver's performance. Limiting the lookahead to only one decision level of $\mathcal{T}_{save}$ was not as good, and looking ahead more than 2 decisions of $\mathcal{T}_{save}$ also degraded performance. This provides some evidence that taking too much control away from the solver and forcing it to make too many decisions from $\mathcal{T}_{save}$ can lead to conflicts that are not as useful to the solver.

Reason quality

The saved trail can be thought of as remembering the solver's recent trajectory. Sometimes we want to follow the past trajectory, but perhaps sometimes we do not. In particular, when adding literals from $\mathcal{T}_{save}$ to $\mathcal{T}$ we can examine the quality of the saved reasons to see if they are worth using. Once we encounter a literal with a low quality saved reason we stop adding literals from $\mathcal{T}_{save}$ to the solver's trail. In particular, we can change lines 33–34 of USESAVEDTRAIL to the following:

31.5 **if** (lowQuality($reason_{save}(l_{save})$)) **break**
32. $\mathcal{T}$.addToEnd(l_{save})
33. $reason(l_{save}) \leftarrow reason_{save}(l_{save})$

Note that the solver will still set the un-added literals as they are unit implied by $\mathcal{T}$, but it might be able to find better reasons for these implicants. There is of course no guarantee that better reasons will be found, but our empirical results show that sometimes this does happen. We experimented with two quality metrics, clause size and clause LBD, obtaining positive results with both. These results also provides evidence against the argument given in the paper by van der Tak et al. [72] that changing literal reasons is not impactful. With an appropriate clause quality metric, the changing of literal reasons can have an impact.

4.5 Enhancements Which Were Not Successful

Learning Multiple Clauses from a Trail

It is possible to learn a second clause from a saved trail if the conflicting literal on the saved trail is different than the conflicting literal on the current trail. If the last decision d the SAT solver made did not come from the saved trail (and assuming that none of the unit propagants of d came from the saved trail), that means when decision d was made, it was possible to make decision $\mathcal{T}_{save}[0]$, which will still be the top of the saved trail, instead. Now, one can scan the propagants on the saved trail that come after $\mathcal{T}_{save}[0]$.

However, we failed to implement this in an efficient way because there are several things to watch out for. One has to ensure that none of the literals that are unit propagants on the current trail after d could originally have come from $\mathcal{T}_{save}$. Because this is too restrictive and also involves bookkeeping that tracks whether each decision has had unit propagants coming from $\mathcal{T}_{save}$ or not, what we tried instead was to allow those unit propagants to come from $\mathcal{T}_{save}$ after restoring $\mathcal{T}_{save}$ back to the state it was in when d was made. However, this was not efficient, and we believe introducing the extra data structure to help restore the trail back to its previous state slowed down unit propagation too much.

Nonetheless, we believe this is a good idea for future exploration if the implementation can be made efficient enough.

Continuing to Use A Trail After Conflict

It is possible to keep using the saved trail even after a conflict is found, but then the reason of every literal you use has to be checked that it is unit under the current trail. In theory, this could still be beneficial since often times a lot of the trail is the same, and many of the literals will still be forced from $\mathcal{T}_{save}$ if their reasons were not made unit as a result of the conflicting literal on $\mathcal{T}_{save}$ (or any of the implications of the conflicting literal).

However, when we tried to implement this, the results of the SAT solver got a lot worse. It is known that the unit propagation process is a hotspot of the SAT solver; it must be kept very fast and modern solvers like cadical rely on the trail and the watchlists of recently accessed literals to be in cache in order to speed things up. Pulling in reasons from literals which are no longer unit is potentially disrupting the cache, and thus the efficiency of the unit propagation process.

4.6 Experiments and Results

In this section we will first describe the setup and benchmarks used for our experiments. We will then discuss a simple analysis of these experiments and the main results of this chapter, which show that trail saving can improve the performance of SAT solvers. We will also discuss how adding enhancements to trail saving made a further improvement over using chronological backtracking alone and trail saving alone.

Next, we will do a finer-grained analysis to give evidence that the reason for the performance improvement is because of a speedup in unit propagation, as hypothesized in Section 4.4.

4.6.1 Setup and Benchmarks

We implemented our techniques in two different SAT solvers, MapleSAT and cadical,[3] both of which have finished at or near the top of SAT competitions for the past several years [95, 84]. We then ran each solver on the 800 total benchmark instances used in the main tracks of the 2018 SAT Competition and 2019 SAT Race. The experiments were executed on a cluster of 2.7 GHz Intel cores with 5000 seconds CPU time and 7 GB memory limits for each instance. We chose not to output or verify the proofs generated by any of the solvers (in order to reduce the resources necessary to conduct the experiments).

4.6.2 Main Experiments

The first set of experiments shows how trail saving and each of the enhancements impacts the performance of SAT solvers and compares it with the impact of chronological backtracking. These are the main results of this chapter that indicate that our techniques were successful. The Par-2 scores[4] obtained and total instances solved by each solver are reported in Figures 4.3, 4.5, and 4.6. We also show the cactus plot of the new version of cadical in Figure 4.4 and MapleLCMDist in Figure 4.7.

[3] Our implementation is available at https://github.com/rgh000/cadical-trail.

[4] The Par-2 score [96] is the sum of all run times of all solved instances plus two times the timeout limit times the number of unsolved instances, all divided by the total number of instances.

	Total	SAT	UNSAT	Avg. Par-2
cadical	532	314	218	4025
cadical-chrono	525	308	217	4086
cadical-trail	529	310	**219**	4060
cadical-trail-multipleBT	530	313	217	3930
cadical-trail-multipleBT-lookahead	531	313	218	4020
cadical-trail-multipleBT-lookahead-reason	**538**	**319**	**219**	**3854**

Figure 4.3: Table of results for cadical (version pulled from github as of January 1, 2020) with non-chronological backtracking, cadical with chronological backtracking, cadical with trail saving, and cadical with trail saving and various enhancements on top. Trail saving without enhancements results in an improvement over chronological backtracking, but not an improvement over the original. Trail saving with all enhancements added results in a significant improvement in number of instances solved and average Par-2 score.

In Figure 4.3 we used the newest version of cadical (downloaded as of January 1, 2020) as the baseline solver, in Figure 4.5 we used the version of cadical published in 2019 [83] as the baseline, and in Figure 4.6 we used MapleLCMDist [97, 98] as the baseline. Each of the baselines were run with standard non-chronological backtracking. We then refer to versions of each solver with additional features implemented on top by adding suffixes. "-chrono" refers to the solver with chronological backtracking enabled (using the solver's default settings), "-trail" refers to the baseline with plain trail saving added (as described in Figure 4.1), "-trail-multipleBT" refers to the baseline with trail saving plus the first enhancement of saving over multiple backtracks, "-trail-multipleBT-lookahead" also adds the enhancement of lookahead for conflicts by 2 decision levels, and "-trail-multipleBT-lookahead-reason" also adds the final enhancement to cease trail saving once a reason of "low quality" is reached. For more details on the enhancements, please see Section 4.2.

Interestingly, chronological backtracking made the newest version of cadical perform worse than the baseline (Figure 4.3). This demonstrates that chronological backtracking is not always beneficial. Trail saving alone did not impact the performance of this solver significantly, but adding all of the enhancements on top of trail saving resulted in solving six more instances and yielding a better Par-2 score than the baseline. The key enhancement for this solver seemed to be the last one where we stop using the saved trail once we detect a reason of "low quality". We tried both clause size and LBD as the clause quality metric, and both yielded a positive gain, with clause size being slightly more effective.

The version of cadical used in Figure 4.5 did show benefits from chronological backtracking in agreement with previously published results [83]. Trail saving alone did not significantly impact this solver, but adding all of the enhancements on top of trail saving resulted in solving the same number of instances as the solver with chronological backtracking did, albeit with a slight increase in the Par-2 score.

MapleLCMDist (in Figure 4.6) is another solver that benefited from chronological backtracking. In this solver trail saving alone solved two more instances than the solver with chronological backtracking did. Adding the first two enhancements on top of trail saving resulted in solving only one more instance but yielded a better Par-2 score than the solver with chronological backtracking. Adding the last enhancement of ceasing trail saving on a "low quality" reason made the performance worse, whether clause size or lbd was used as the clause quality metric. This suggests that the enhancements to trail saving have different impacts on different solvers.

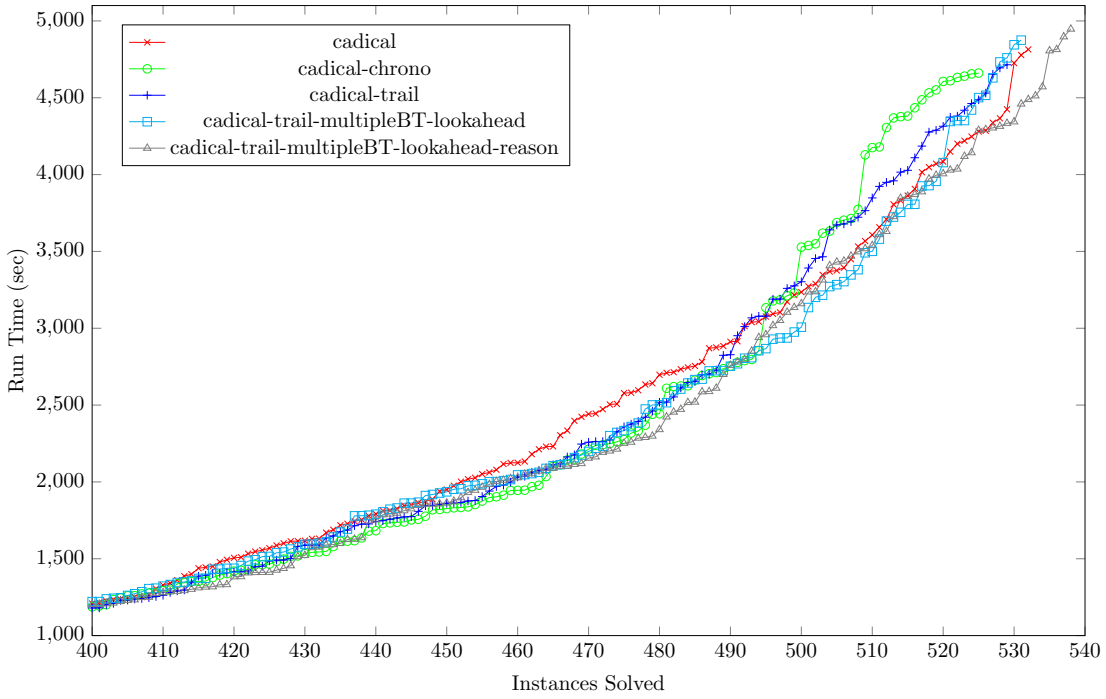

Figure 4.4: Cactus plot for the newest version of cadical comparing standard non-chronological backtracking to chronological backtracking and various configurations of trail saving. The first 400 problems were solved in less than 1200 seconds, so that part of the plot is truncated. cadical with trail saving and all enhancements solves instances faster and solves more instances overall.

	Total	SAT	UNSAT	Avg. Par-2
cadical-19	492	289	203	4378
cadical-19-chrono	**498**	**292**	**206**	**4300**
cadical-19-trail	493	290	203	4403
cadical-19-trail-multipleBT-lookahead	496	**292**	204	4356
cadical-19-trail-multipleBT-lookahead-reason	**498**	**292**	**206**	4351

Figure 4.5: Table of results for an earlier version of cadical as it existed at the time of the paper by Möhle et al. [83], cadical with its first implementation of chronological backtracking (named "chrono" when it was originally published [83]), and various configurations of trail saving. We call this cadical-19 to distinguish it from the newer version of cadical used in other experiments. Although chronological backtracking shows significant improvement, that improvement is matched by trail saving with all enhancements.

	Total	SAT	UNSAT	Avg. Par-2
maple	458	259	199	4746
maple-chrono	470	271	199	4613
maple-trail	**472**	271	**201**	4618
maple-trail-multipleBT-lookahead	471	**272**	199	**4597**
maple-trail-multipleBT-lookahead-reason	469	271	198	4633

Figure 4.6: Table of results for MapleLCMDist with standard backtracking, with chronological backtracking, and with various configurations of trail saving. Trail saving provides an improvement in instances solved and speed (as measured by Par-2), but the enhancements make the performance worse for MapleLCMDist.

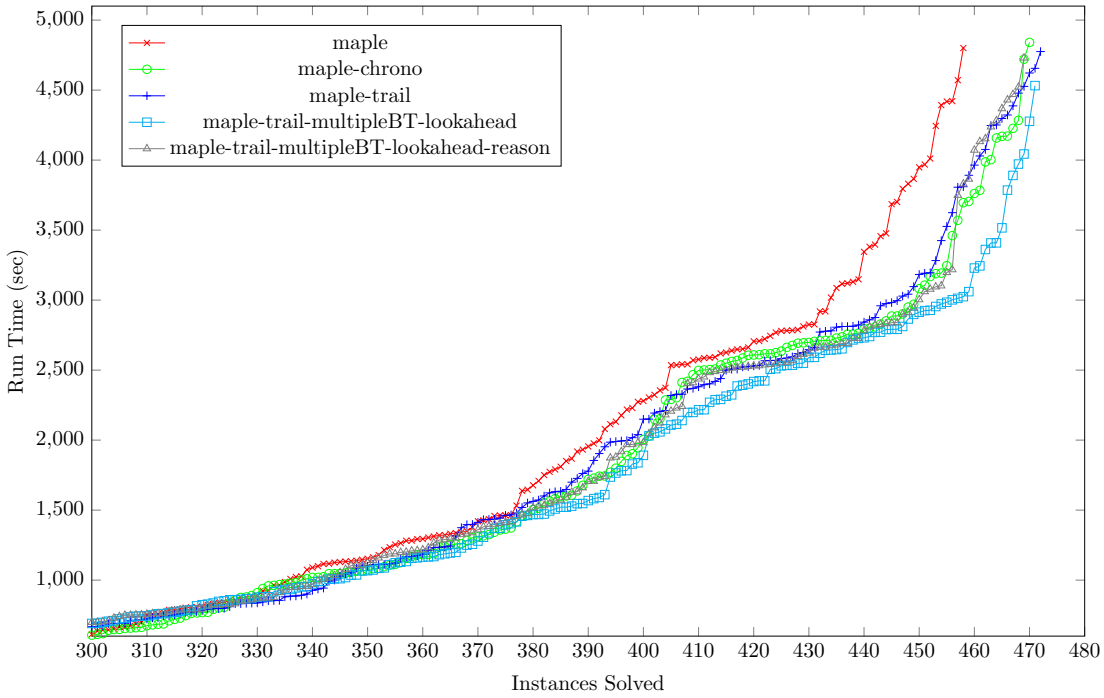

Figure 4.7: Cactus plot for MapleLCMDist comparing standard non-chronological backtracking to chronological backtracking and various configurations of trail saving. The first 300 problems were solved in less than 600 seconds, so that part of the plot is truncated. Trail saving solves slightly more instances, while trail saving with the lookahead enhancement is solving instances faster, although not solving more instances overall.

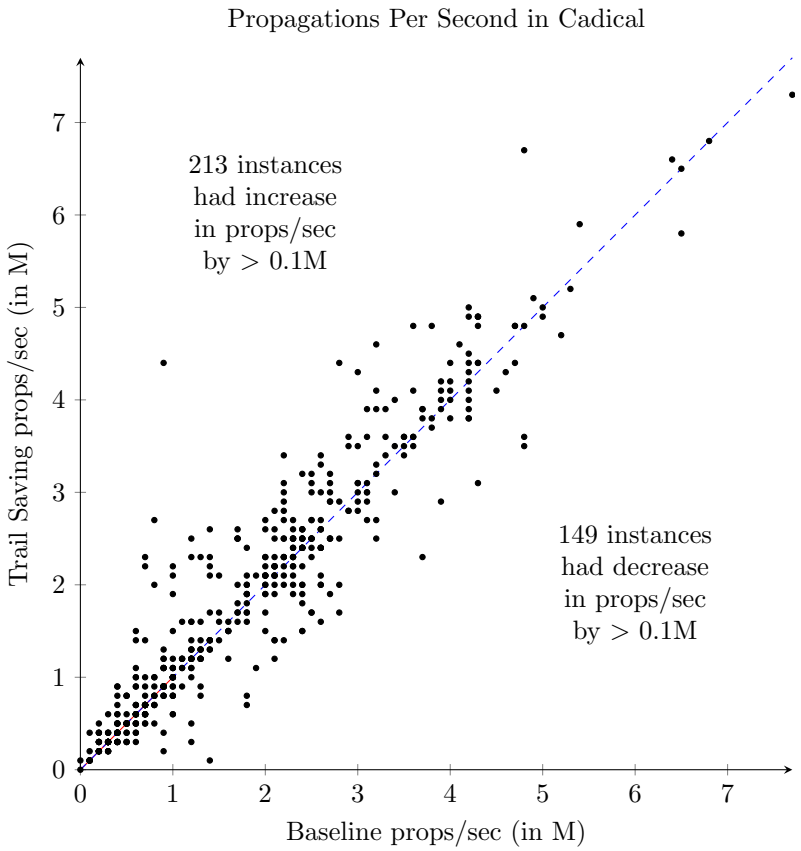

Figure 4.8: Comparing unit propagations per second performed by the SAT solver with and without trail saving in cadical. Dots on the left hand side of the diagonal line represent instances where trail saving had more propagations per second, and dots on the right hand side of the diagonal line represent instances where trail saving had fewer propagations per second. We can see that the instances skew towards the left at least slightly (thus indicating trail saving tends to increase propagations per second).

4.6.3 Measuring Propagations Per Second

As a follow-up to the experiments above, we did a more fine-grained analysis to give further evidence that the improvement trail saving provides really comes from speeding up the propagation, as opposed to something more subtle but unintended such as changing the order of the watchlists. Although the latter may seem unlikely, when dealing with SAT solvers, it is very difficult to determine the exact reason that a technique is improving things, because sometimes a seemingly minor change can unexpectedly cause a significant change in performance.

In Figure 4.8 we plotted the propagations per second for each instance with and without trail saving enabled in cadical (rounded to the nearest 0.1 million). This data was computed from the same experiments run before. Only the 506 instances solved by both solvers are plotted. With trail saving enabled, 213 instances had more propagations per second, 149 instances had fewer propagations per second, and 144 instances had about the same number of propagations per second. On the graph, points above the diagonal represent instances where trail saving had faster propagation. The graph illustrates that the majority of the time cadical with trail saving enabled propagated literals faster than cadical without trail saving enabled, as we predicted would be the case earlier in this chapter.

4.7 Discussion

In this chapter, we identified some redundant work that SAT solvers perform, namely that they reconstruct the same portions of the trail over and over again, and have introduced a new technique, trail saving, to efficiently reuse the assignment trail in a SAT solver. There is another new SAT solving technique, chronological backtracking, that also takes care of similar redundant work that trail saving takes care of (and does so with less overhead); however, we argued that chronological backtracking can have other undesirable effects on the execution of the main SAT solving search that trail saving can avoid.

We have demonstrated experimentally that our trail saving technique can speed up two state-of-the-art SAT solvers, cadical and MapleSAT, as effectively or more effectively than chronological backtracking can on a wide range of benchmarks from the SAT Competitions. We then showed empirically that trail saving leads to faster unit propagation (by measuring the number of unit propagations per second), confirming our hypothesis and helping to explain why it is effective.

We also introduced three enhancements one can implement when using a saved trail, including saving the trail over multiple backtracks and appending them together, using the saved trail to lookahead for conflicts, and rejecting the usage of the trail's reason when the reason is of "poor quality". We demonstrated experimentally that these enhancements can sometimes improve a SAT solver's performance. We have proven that trail saving and all enhancements we proposed are sound. Some other possible enhancements were also proposed and tried, but were not successful for us.

4.7.1 Future Work

There are many avenues that can be pursued in future work. One interesting direction would be to use the saved trail to help make inprocessing techniques [92] faster. Especially for those inprocessing techniques that rely on doing unit propagation to gather information (sometimes referred to as probing), a saved trail can be thought of as a cheap rough approximation to unit propagation.

Using the saved trail to learn multiple clauses from a single conflict is also possible when the saved trail and current trail each have a different contradictory literal. Our attempts at implementing this did not lead to success, but there are alternative implementations to try. Learning multiple clauses could also be possible when saving multiple different trails at a time instead of just one (e.g., when the solver has backtracked to the same spot in the search tree more than once).

It is also possible to combine trail saving with chronological backtracking, but it would require further work to determine whether or not this would be useful and how to best approach it. Our attempts to do so did not yield positive results. Recently in MergeSAT [99], a combination of the two has been implemented, but an extensive analysis of whether the combination helps on a wide range of benchmarks is still to be done.

Finally, it may be possible to extend trail saving to other domains. In a paper by Bankovic and Scepanovic [23], trail saving was extended to SMT solving with mixed success, but left room for the implementation to be developed further. Other domains which perform propagations to construct partial assignments, and that tend to have repeating sequences of assignments during search, would be good candidates. We think constraint satisfaction (CSP) solving and model counting (#SAT) would be especially interesting to try.

4.8 Application: Trail saving on Planning Instances

Trail saving seems to work especially well on planning instances, even though we didn't restrict any of our experiments in the last section to focus specifically on planning. A version of the cadical SAT solver implementing trail saving [100] won the Planning Track of the 2020 SAT Competition [96].

Planning instances were one of the first instances that SAT solvers were used for and have always been of interest to the SAT community. There is an extensive literature on techniques for efficiently solving planning instances with SAT solvers [101].

4.8.1 Objective

To further demonstrate that trail saving is effective on planning instances, we implemented trail saving into a SAT solver and tested how well it did in the FOND-SAT [102] planner.

FOND-SAT is an efficient SAT-based planner for fully observable non-deterministic (FOND) problems. Heuristic search planners such as PRP [103] tend to dominate SAT-based planners on FOND planning domains, including most of those seen in the International Planning Competition [104]. However, a paper by Geffner et al. [102] introduced some interesting new domains where FOND-SAT dominates.

Our objective is to show that trail saving can be effective in this domain, but first we will have to upgrade the SAT solver that FOND-SAT is using to a more modern SAT solver. FOND-SAT used minisat, a good but old solver, so our first task was to make FOND-SAT use cadical.

4.8.2 Experimental Setup

For all experiments listed in this section, each planner was run on a 4.2 GHz CPU. Each instance was given a time limit of 3600 seconds and a memory limit of 4 GB. The same 484 total instances over 14 different FOND planning domains from the paper by Geffner et al. [102] were used for each experiment. The results focus on finding strong cyclic plans in all experiments in this section. The source code for the best version of FOND-SAT incorporating all of the experiments in this section can be found at https://github.com/rgh000/FOND-SAT-speedup.

4.8.3 Trying Different SAT Solvers

The original version of FOND-SAT used minisat [105], which does not include many of the SAT solving techniques developed in the last 15 years. Some of these heuristics include new clause deletion heuristics, learnt clause minimization, new branching heuristics, phase targeting based on local search, and others [106].

The first attempt to improve the performance of FOND-SAT was to experiment with more modern SAT solvers in place of minisat. Some of the solvers tested were cadical [107], Glucose [108], and MapleSAT [109]. The results of the original, minisat, and the best SAT solver, cadical, are listed in Figure 4.9 alongside the results of PRP for reference.

It can be seen that cadical solves more problems than minisat in most of the domains and is quite a bit faster, however cadical fails to close the gap between FOND-SAT and PRP with any significance in those same domains that PRP dominates in. In general, using a better SAT solver will make FOND-SAT solve slightly more problems and speed up quite a bit, but it does not seem to be the answer to significantly improving FOND-SAT's performance in domains it previously had trouble with.

Instance Domain	minisat solved	minisat time(s)	cadical solved	cadical time(s)	PRP solved	PRP time(s)
tireworld-truck	73	331.1	**74**	207.0	21	177.3
triangle-tireworld	3	214.1	4	219.0	**38**	59.1
islands	57	317.4	**57**	125.8	28	45.4
zenotravel	5	348.4	5	443.5	**14**	4.6
acrobatics	3	24.0	4	161.5	**8**	7.9
earth_observation	6	105.9	7	859.8	**40**	0.4
doors	14	637.3	**15**	223.7	12	1.3
first-responders-08	49	248.3	51	232.3	**74**	0.5
miner	48	616.0	**50**	639.5	10	239.6
elevators	7	101.9	7	418.9	**14**	0.2
beam-walk	3	421.8	3	106.7	**11**	10.8
spiky-tireworld	10	1039.7	**10**	517.2	1	32.4
tireworld	12	17.3	**12**	15.9	11	0.2
blocksworld-08	10	86.1	10	433.7	**29**	1.1
total	300		309		311	

Figure 4.9: Number of instances solved followed by total average time taken per instance (in seconds) for minisat, cadical, and PRP.

4.8.4 Trying Different Heuristics

Tuning heuristics has been successful in improving the performance of SAT-based classical planners [110]. The next attempt to improve the performance of FOND-SAT was to try several different heuristics and determine which ones work best for FOND planning.

Several different variable selection heuristics were experimented with, including prioritizing action variables and prioritizing action variables which have a fluent in common with the goal state, but none of these yielded positive results. Similarly, many clause deletion heuristics were tried such as prioritizing shorter clauses more than usual, but none of these ended up being advantageous either.

The first heuristic that made a positive difference was to add in trail saving, as covered earlier in this chapter. The other heuristics that did make a positive difference in performance were heuristics that turn off features in modern SAT solvers which target satisfiable instances. This makes sense because the FOND-SAT algorithm often has to send many unsatisfiable instances to the SAT solver before it sends a satisfiable instance. The heuristics that were turned off in the best version were chronological backtracking [111], stabilized restarts [112], and random walk exploration [111]. This version beat the others in total problems solved (or total run time in the case of a tie) on every domain except *islands*. The results comparing various different heuristic settings are listed in Figure 4.10.

4.8.5 Observations

Altering the SAT solver inside of FOND-SAT made improvements in speed and modest improvements in total number of problems solved. Furthermore, we saw that similar to regular planning instances, trail saving also makes a modest but definite improvement in performance for FOND instances that have been translated into SAT (at least for the FOND-SAT tool). However, FOND-SAT still can not compete with PRP on most of the FOND planning domains that come from the planning competitions, despite having been given a significant performance improvement. Given the fact that the biggest hindrance for the SAT-based approach to FOND

Instance Domain	default solved	default time(s)	trail-sav solved	trail-sav time(s)	unsat solved	unsat time(s)
tireworld-truck	74	274.0	74	238.3	74	207.0
triangle-tireworld	4	570.2	4	549.2	4	219.0
islands	57	209.4	55	165.0	55	125.8
zenotravel	5	714.1	5	638.8	5	443.5
acrobatics	4	272.7	4	272.4	4	161.5
earth_observation	7	614.5	8	786.8	10	859.8
doors	15	540.9	15	406.8	15	223.7
first-responders-08	51	280.2	52	306.8	52	232.3
miner	50	755.2	50	713.9	50	639.5
elevators	7	73.9	7	71.9	8	418.9
beam-walk	3	216.3	3	214.9	3	106.7
spiky-tireworld	10	794.5	10	699.9	10	517.2
tireworld	12	38.0	12	43.0	12	15.9
blocksworld-08	10	92.4	11	397.2	12	433.7
total	309		310		314	

Figure 4.10: Number of instances solved and average run times for the default cadical solver, the solver with trail saving (trail-sav), and the solver with trail saving, without stabilized restarts, and without random walk exploration (unsat).

planning is the encoding size of certain problems, it is likely that a radical modification to the encoding is required to compete with PRP in the problem domains that PRP currently dominates in.

Chapter 5

Large Neighbourhood Search for Anytime MaxSAT Solving

5.1 Introduction

5.1.1 Motivation

Due to significant advances in maximum satisfiability (MaxSAT) solvers over the past decade, MaxSAT [113, 114] has proved itself to be very effective at modelling and solving a range of discrete optimization problems (see the chapter on maximum satisfiability [114] in the Handbook of Satisfiability).

Although the field had previously concentrated on building better **complete MaxSAT solvers**, i.e., solvers designed to find optimal solutions, in recent years **incomplete MaxSAT solvers** have become increasingly important. Incomplete solvers do not aim to find optimal solutions, but rather at finding good solutions in less time. In particular, they offer no guarantees on the quality of the solutions they produce, but can be more readily applied in **anytime contexts**. As the range of applications of MaxSAT solving has grown, some real-world applications, e.g., physical design stage for Computer-Aided Design [40], high school timetabling [115], etc., have been found to generate problems that are either too large or too difficult for complete solvers to solve to optimality in a timely manner. Hence, as with many real world applications of optimization technology, good but not necessarily optimal solutions produced in a reasonable time frame are sought.

Driven by this need, several new ideas, e.g., [40, 116, 117] have been developed and proved successful in improving incomplete MaxSAT solvers (as demonstrated in the annual MaxSAT evaluations[1]), and anytime MaxSAT solving has become an increasingly active area of research in the past few years.

Many of the best anytime MaxSAT solvers rely on taking a solution and continually improving it slightly, by either guiding it towards satisfying more weight (i.e., equivalently falsifying less cost) with polarity selection heuristics [40], adding constraints that prevent the next solution from being worse than the last one [77], or flipping the truth value of some variables to search for a better solution [118]. The aforementioned anytime algorithms all have a similar property in that they search for new solutions that are "near" the last solution found. What we mean by "near" the last solution is that they either will share a lot of the same truth assignment values or they will satisfy almost all of the same clauses. Indeed, when we observe the behaviour of these algorithms, we see that they all tend to find high quality MaxSAT solutions very quickly (to a point

[1] https://maxsat-evaluations.github.io/2021

that is usually a high percentage of the optimal), but then "stall" at a local optima and have a hard time improving the solution past a certain threshold. This is because most of them have a local search (or locally optimizing) component and lead to a lot of redundant work being performed as the anytime solver continues to find the exact same solutions over and over again. Although incremental SAT solvers can alleviate some of this redundant work, they do not alleviate all of it and it would be better if we could search through these similar solutions in fewer iterations to find the best one. Our idea is to do a more thorough and complete search of a local neighbourhood of the solution space in order to find the optimal solution in that local neighbourhood all at once. The Large Neighbourhood Search algorithmic framework is a perfect candidate to help us accomplish this.

Large Neighbourhood Search (LNS) [19] is a well-known technique for incomplete solving which has been successfully applied to many optimization problems. The LNS framework involves repeatedly taking an initial solution S, defining a "large" neighbourhood N of S, and then searching N for a better solution than S. Often N is small enough (when compared to the original problem) that it can be exhaustively searched with a complete solver. LNS's effectiveness at finding good solutions is thus determined by the initial solution S and by the selection of the neighbourhood N. In particular, in LNS one aims to select an N that is likely to contain better solutions and that can be effectively searched. Typically, multiple different initial solutions S, and for each S multiple different neighbourhoods N, are tried. LNS has been successfully applied in a number of different domains [119, 120], showing that it is often possible to develop effective methods for selecting S and N. *One of the main contributions of this chapter is to use LNS to develop an algorithm for anytime MaxSAT which improves the state-of-the-art solvers.*

5.1.2 Summary of Our Contributions

Interestingly, although there has been some related work (as discussed later in this chapter), to the best of our knowledge, LNS has not yet been directly applied in incomplete MaxSAT solving. As pointed out above, LNS offers the possibility of exploiting complete solvers in the context of incomplete solving. Since complete MaxSAT solvers have seen tremendous advances over the past decade, investigating the use of LNS in incomplete MaxSAT solving seems overdue. In this chapter we address that gap and develop a few potential neighbourhood selection policies, before settling on one neighbourhood selection policy that gives the best empirical performance.

Although our Large Neighbourhood Search algorithm does not completely eliminate the redundancy of finding the exact same solutions over and over again, since it is technically possible to select the exact same neighbourhood twice (and also possible to select a subset of a previous neighbourhood), we would argue that it is highly unlikely that the same neighbourhood would be selected twice because we are doing a probabilistic selection among literals and have fixed set sizes that are almost always thousands of literals in size.[2] Our Large Neighbourhood Search for MaxSAT algorithm is also designed to reduce some redundancy that locally optimizing solvers suffer from where they stay in the same local region for too long. By exhaustively searching a neighbourhood with a complete solver and moving on, we believe we are avoiding staying in the same local region for too long. Of course, many of these locally optimizing algorithms have their own remedies for escaping local minima, so we are not making the claim that our algorithm definitely does less redundant work than a local search algorithm, but we think it is very likely that it does.

We then do a more in-depth study showing that our approach can often improve very high quality solutions

[2]It is worth noting that we also did experiments where we more explicitly tried to rule out selecting the exact same neighbourhood multiple times with the use of a Bloom filter [121], and the performance was almost identical).

found by other incomplete solvers faster than those incomplete solvers can improve them themselves, but also find that our approach is less effective at turning poor quality solutions found by those solvers into good solutions. Using this insight we are able to improve the state-of-the-art in incomplete MaxSAT solving by budgeting some time to run our LNS approach on the solutions produced by another incomplete solver. Additionally, LNS finds brand new better solutions for some of the instances that none of the other anytime solvers were able to find. Our method works for both weighted and unweighted instances, and opens the door for investigating other potential ways LNS can be used to improve incomplete MaxSAT solving.

5.1.3 Chapter Outline

In the remainder of this chapter, we will first introduce some necessary background and notation before going on to describe a generic way to apply LNS to MaxSAT solving. We will then describe exactly how we applied LNS to MaxSAT and give our neighbourhood selection policy, which is the key to getting good performance. We will also discuss some other possible neighbourhood selection policices, including several that we tried and did not have as much success with.

Finally, we will present experiments designed to compare how well LNS improves individual solutions, as well as experiments showing how our LNS approach improved three state-of-the-art anytime MaxSAT solvers for instances which run in about 300 seconds or less.

5.2 Background

We assume the reader is familiar with MaxSAT solvers and conflict-driven clause learning (CDCL) SAT solvers [69]. Please refer to Chapter 2 of this thesis and Chapter 4 of the Handbook of Satisfiability [70] for more detailed explanations. Below we will introduce some more specific notation and definitions that will be useful for the remainder of this chapter.

5.2.1 Assignments and Solutions for MaxSAT

A MaxSAT instance F is specified by a **weighted CNF**. A CNF is a Boolean formula expressed as a conjunction of **clauses**, where each clause is a disjunction of **literals**, and each literal is a Boolean variable v or its negation $\neg v$. The CNF is weighted if every clause has an associated weight that is a positive number or infinity. The **soft clauses** of the instance F, $soft(F)$, are those with finite weight while those with infinite weight are the **hard clauses**, $hard(F)$. We use c_w to denote the weight of clause c, and we often regard F to be a set of clauses (understood to be a conjunction), and a clause to be a set of literals (understood to be a disjunction).

A **total truth assignment** π for instance F is a set of literals containing exactly one literal of every variable of F. These literals specify the truth assignments given to the variables: $\pi(v) = true$ when $v \in \pi$ and $\pi(v) = false$ when $\neg v \in \pi$. π **satisfies** a clause c, written as $\pi \models c$, if it contains at least one literal of c. Thus, an empty clause can never be satisfied. The cost of π, $cost(\pi)$, is the sum of the costs of the clauses it falsifies (i.e., clauses c such that $\pi \not\models c$): $cost(\pi) = \sum_{c \in \{\pi \not\models c\}} c_w$. π is a **feasible solution** of F if it satisfies every hard clause of F (equivalently if it has finite cost). π is an **optimal solution** if it is feasible and has lowest cost amongst all feasible solutions. That is, an optimal solution satisfies all hard clauses and falsifies a minimum weight of soft clauses (equivalently, it satisfies a **maximum** weight of soft clauses).

Complete and incomplete MaxSAT solvers are used to find feasible solutions. **Complete solvers** aim to find optimal solutions while **incomplete solvers** aim to more rapidly find low cost solutions without any general guarantees on quality. **Anytime** MaxSAT solving aims to find the lowest cost solution possible within a given time limit.

A **partial truth assignment** α is a set of literals with at most one literal of any variable (but not always containing a literal for every variable). The **reduction** of F by a partial truth assignment α, written as $F|_\alpha$ is the process of removing from F all clauses containing a literal of α and then removing from the remaining clauses the negation of all literals in α. The weights of the unremoved clauses are unchanged. This process yields a new instance that is logically equivalent to the formula $F \wedge \bigwedge_{\ell \in \alpha} \ell$. Note, that $F|_\alpha$ is smaller than F and is typically easier to solve—especially if α contains a large number of literals. $F|_\alpha$ might contain empty clauses due to the second step of the reduction process. If it contains an empty hard clause, it has no feasible solutions. If it contains empty soft clauses, the weight of those soft clauses become part of the $cost$ of every total truth assignment of $F|_\alpha$.

5.2.2 Large Neighbourhood Search

The LNS algorithm takes an initial suboptimal solution S, keeps some part of the formula **fixed** and relaxes (or reduces) the rest of S to form $S_{relaxed}$, then re-optimizes $S_{relaxed}$. The solution from $S_{relaxed}$ is then combined back with the fixed part of the solution to form a new solution, hopefully better than S. The set of solutions which can be obtained during this re-optimization is called the **neighbourhood**. When we refer to a **neighbourhood selection policy**, we are referring to the procedure used to relax the problem, and thus determine the neighbourhood to explore.

Although for our purposes the fixing and relaxing of solutions involves deciding whether to keep certain variables' truth values fixed or not, LNS generally is a higher level paradigm and can fix or relax other parts of a solution as well. One example we explore later is the fixing of clauses from soft to hard, which was not as successful for us, but there might be other ways of fixing parts of MaxSAT solutions that we did not think of.

5.3 Using LNS to Solve MaxSAT Problems

In this section, we will lay out our LNS approach for MaxSAT, which we consider to be a generic way to apply LNS to MaxSAT. We will then provide pseudocode for our method and get into some of the finer details of our implementation. Next, we will outline our precise neighbourhood selection policy, the key to getting good performance for LNS. Lastly, we will talk about some other neighbourhood selection policies and heuristics we tried that did not work.

5.3.1 Our Approach

The first step in applying the LNS technique to a MaxSAT problem is to obtain an initial suboptimal solution to the problem. For this, we ran an anytime MaxSAT solver for a fixed amount of time and used the best solution it found as our initial solution. The next step is to select a neighbourhood and reduce the problem. We did this by selecting a subset of the initial solution assignment and reducing the MaxSAT problem by the partial assignment that corresponds to that subset.

The next step is to solve the reduced subproblem. For this, we sent the reduced subproblem to a complete MaxSAT solver and solved it optimally. We will refer to the process of selecting a neighbourhood and solving the corresponding reduced subproblem as one round of LNS.

The last step is to take the best solution from our most recent round of LNS and use it as the initial solution to perform another round of LNS. We continue to perform rounds of LNS to keep trying to obtain better solutions until we are out of time.

Implementation

An outline of our approach to using LNS in MaxSAT is given in Algorithm 2. Our approach is parameterized by two auxiliary solvers: (1) an anytime solver AT_MaxSlv that provides an initial feasible solution for the instance F and (2) a complete solver C_MaxSlv which is able to efficiently and exhaustively search a selected neighbourhood of the current feasible solution. AT_MaxSlv is called first to obtain a feasible solution π (a total assignment). In our implementations, we ran AT_MaxSlv for a fixed amount of time and used the best solution it found as our initial solution; we experimented with different anytime solvers but always used the same complete solver MaxHS [122].[3]

We keep track of the best solution found in the variable $best$, initializing this to π, the solution returned by AT_MaxSlv. The next step is to select a neighbourhood of π than can be searched for a better solution. A common way to do this is to fix some subset of the variables to the value provided in π and leave the remaining variables unfixed. The different assignments to the unfixed variables define the neighbourhood.

In our approach the initial solution π is a total assignment and we can select literals of π to form a partial assignment α whose size is approximately a factor $\delta\pi$ ($\delta < 1$). (Propagation, described below, makes it impossible to select an α with an exact size). The neighbourhood N is thus defined by the different assignments to the variables not appearing in α, while the variables in α remain fixed.

Proposition 2 *The best solution of F when α is fixed is $\alpha \cup N\_best$, where N_best is a solution to $F|_\alpha$.*

Proof It is not difficult to see that N is exactly the set of feasible solutions of $F|_\alpha$. In particular, $F|_\alpha \equiv F \wedge \bigwedge_{\ell \in \alpha} \ell$. Hence, β is a feasible solution of $F|_\alpha$ iff $\alpha \cup \beta$ is a feasible solution of F. Thus, by finding an optimal solution N_best of $F|_\alpha$ we find the best solution of F that extends α which is $\alpha \cup N\_best$. ■

The benefit of Proposition 2 is that $F|_\alpha$ is simply a new MaxSAT instance. If α is large enough (and thus $F|_\alpha$ is small enough), $F|_\alpha$ can be efficiently solved to optimality by C_MaxSlv.

We refer to the set of literals in the selected partial assignment α as the **fixed set**, and the process of selecting a fixed set α and then solving $F|_\alpha$ as one **round** of LNS.

Note that for π computed on line 8 of Algorithm 2 we must have that $cost(\pi) \leq cost(best)$, unless C_MaxSlv on line 7 was interrupted. This is because the truth assignments given in $best$ to the variables of $F|_\alpha$ form a feasible solution of $F|_\alpha$. So N_best (an optimal solution of $F|_\alpha$) cannot have cost greater than the cost of this feasible solution. That is, at worst C_MaxSlv will return an N_best such that $cost(\pi) = cost(\alpha \cup N\_best) = cost(best)$. In these cases $best$ will be unchanged and Algorithm 2 will select a different neighbourhood to try to optimize next. When, however, a better solution than $best$ is found, $best$ will be updated and Algorithm 2 will start selecting neighbourhoods of this new solution.

An example of the LNS process outlined above is illustrated in Example 12.

[3] In recent MaxSAT evaluations MaxHS has consistently been one of the best complete solvers available [123]. In future work the effect of switching complete solvers will be investigated.

Algorithm 2: Generic LNS algorithm for MaxSAT

```
1
2  Input: weighted CNF F, initial solution solver provider AT_MaxSlv, complete MaxSAT solver C_MaxSlv
3  π = AT_MaxSlv(F)                                   /* for fixed time bound */
4  best = π
5  while time left do
6  |   randomly select α ⊂ π of size ≈ δ|π|           /* fixed set */
7  |   N_best = C_MaxSlv(F|_α)                        /* interrupt if out of time */
8  |   π = α ∪ N_best
9  |   best = (cost(π) < cost(best)) ? π : best
10 return best
```

Example 12 *Suppose for a MaxSAT problem F with hard clauses* $H(F) = \{\{x_1 \vee x_2 \vee x_3\}, \{x_1 \vee \neg x_2 \vee \neg x_3\}$ *and soft clauses* $S(F) = \{c1 = \{\neg x_1\}, c2 = \{\neg x_2\}, c3 = \{\neg x_3\}\}$ *with* $cost(c1) = 4$, $cost(c2) = 3$, *and* $cost(c3) = 2$ *that the last solution found was* π *where* $\pi(x_1) = false, \pi(x_2) = true, \pi(x_3) = false$ *and, thus,* $cost(\pi) = 3$. *Now suppose that the fixed set selected from this solution is* $\alpha = \{x_1\}$, *meaning* $\pi(x_1) = false$ *stays fixed. Next, we compute* $F|_\alpha$, *which leaves* $H(F|_\alpha) = \{\{x_2 \vee x_3\}, \{\neg x_2 \vee \neg x_3\}\}$ *and* $S(F|_\alpha) = \{c_2, c_3\}$. *After sending* $F|_\alpha$ *to a maxsat solver, it will return the optimal solution to the reduced problem, which is* $N\_best = x_2 = false, x_3 = true$, *and the new assignment will be* π_{new} *such that* $\pi_{new}(x_1) = false, \pi_{new}(x_2) = false, \pi_{new}(x_3) = true$, *where* $cost(\pi_{new}) = 2$. *From then on, the current solution the algorithm will work to improve will be* π_{new}.

Note that because we were restricted to the neighbourhood of solutions where $\pi(x_1) = false$, *other feasible solutions to F where* $\pi(x_1) = true$ *were not considered by the maxsat solver, e.g., the solution* π_{bad} *where* $\pi_{bad}(x_1) = true, \pi_{bad}(x_2) = true, \pi_{bad}(x_3) = true$ *and* $cost(\pi_{bad}) = 9$ *was not considered.*

Figure 5.1 gives a pictorial representation of what happens when we are executing a round of LNS. We are restricting ourselves to searching in a neighbourhood, N, which is often much smaller than the entire solution space, S. Assignments which are explored by C_MaxSlv in our algorithm will only be "close" to the initial solution, π_i, i.e. they will both share all of the same assignments as in α. Note that since we are using a MaxSAT solver, we will get a feasible solution returned (i.e., either π_1, π_2, or π_i) instead of an assignment which is not a solution (e.g., π_z). Also, since our implementation uses a complete MaxSAT solver for C_MaxSlv, we are guaranteed to get the lowest cost solution in N.

5.3.2 Neighbourhood Selection

The selection of the neighbourhood N is key to the performance of LNS and several strategies have been explored in other contexts, such as in a paper by Lombardi and Schaus [124]. We tried various methods for selecting N and found that the following popular probabilistic method worked best in our experiments.

The method is simply to randomly select a subset of the literals of the current feasible solution π as the fixed set α, and define N to be the varying assignments to the unfixed variables not in α. The random selection is done using a biased distribution over the literals in which literals we would prefer to keep fixed are given higher probability. Using biased probability distributions to select neighbourhoods has been done in other contexts, such as in a paper by Parragh and Schmid [125]. We similarly found that a biased distribution was much more effective than selecting uniformly at random.

The probability distribution that we found to work best was to first assign a weight to each literal $\ell \in \pi$ equal to the total weight of the soft clauses containing it (these clauses are satisfied by π since π contains

Pictorial Representation of LNS for MaxSAT

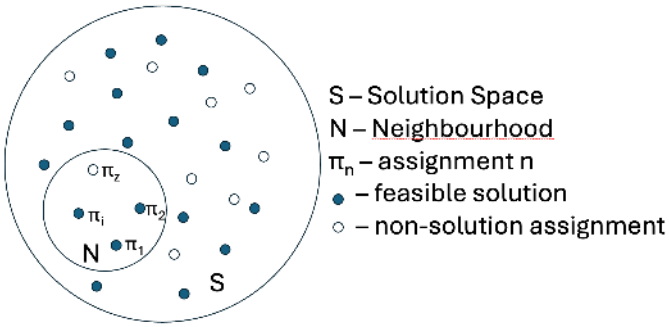

Figure 5.1: This diagram conceptually represents what is happening during a round of LNS. The space of all solutions, S, is much larger than the neighbourhood of solutions, N, we are restricting to. We are only searching for solutions which are conceptually "near" the initial solution, π_i, and we will only ever return a feasible solution (we will not return, e.g., π_z).

ℓ) minus the total cost of falsified soft clauses containing $\neg\ell$, where π is used to determine whether or not a clause is falsified. Normalizing these weights yields the required probability distribution over the literals.

Hard clauses were ignored for our weighting, and a slight improvement was obtained by making the weighting exponential. To be precise, in our final implementation the weight ℓ^w assigned to a literal $\ell \in \pi$ is

$$\ell^w = 2^{\sum\limits_{c \in soft(F)} \begin{cases} c_w, & \text{if } \ell \in c \\ -c_w, & \text{if } \neg\ell \in c \text{ and } \pi \not\models c \\ 0, & \text{otherwise} \end{cases}},$$

and these weights are then normalized to form a distribution over the literals in π, and finally α, a subset of π, is randomly selected from this distribution (selecting without replacement) to form the set of fixed literals. For efficient weighted random sampling, we used the algorithm described in a paper by Efraimidis and Spirakis [126].

These weights capture the intuition that we would like to keep literals of π fixed when they help to satisfy a large weight of soft clauses, and when flipping its value satisfies only a small weight of soft clauses. Using exponential weights amplifies the difference between the literals satisfying (falsifying) slightly different weights of soft clauses, but using probabilities and random selection means that literals with high score might not be selected and thus are allowed to vary in the selected neighbourhood. Example 13 illustrates how to calculate weights for each variable for a particular example.

Example 13 *Suppose for a MaxSAT problem F with hard clauses* $H(F) = \{\{x_1 \vee x_2 \vee x_3\}, \{x_1 \vee \neg x_2 \vee \neg x_3\}$ *and soft clauses* $S(F) = \{c_1 = \{\neg x_1\}, c_2 = \{\neg x_2\}, c_3 = \{\neg x_3\}, c_4 = \{x_1 \vee x_2\}, c_5 = \{x_2 \vee x_3\}\}$ *with* $cost(c_1) = 4$*,* $cost(c_2) = 3$*,* $cost(c_3) = 2$*,* $cost(c_4) = 1$*, and* $cost(c_5) = 1$ *that the last solution found was* π

where $\pi(x_1) = false, \pi(x_2) = true, \pi(x_3) = false$ and, thus, $cost(\pi) = 3$. Then, $\neg x_1^w = 2^4 = 16$, since $\neg x_1 \in c_1$ and $\pi \models c_1$. $x_2^w = 2^{1+1-3} = \frac{1}{2}$, since $x_2 \in c_4$, $x_2 \in c_5$ and $\pi \models c_4$, $pi \models c_5$ while $\neg x_2 \in c_2$ and $\neg x_2 \not\models c_2$. $\neg x_3^w = 2^2 = 4$ since $\neg x_3 \in c_3$ and $\pi \models c_3$. $\neg x_1^w, x_2^w, \neg x_3^w$ would then be used as weights during the weighted random selection of the fixed set (where a higher weight means a higher chance of the corresponding variable being selected to be fixed).

5.3.3 Other Minor Improvements

Propagation has been used in other contexts when selecting neighbourhoods for LNS [127], and we found it to be beneficial in our approach. In particular, as we randomly select each literal ℓ to be fixed, we incrementally add ℓ to $hard(F)$ and perform unit propagation on $hard(F)$. All unit implied literals are forced by the hard clauses to be fixed in all feasible solutions containing the previously randomly selected set of fixed literals. Hence, we add these unit implied literals to the fixed set α and continue to randomly select literals for α amongst the remaining literals of π not yet in α, stopping when α reaches or exceeds the desired size, $\delta|\pi|$ ($\delta < 1$). (Since we cannot predict how many unit implicants will be added to α we cannot select an exactly sized α.)

Proposition 3 *Unit propagating all literals selected so far in the fixed set α, and adding all of those new unit propagants to α, still leaves $\alpha \subset \pi$.*

Proof Note that since π is a feasible solution satisfying $hard(F)$ and all selected literals are in π, all unit implied literals u found in this way must also be in π. Hence, we still have that $\alpha \subset \pi$. ■

One benefit of propagation is that it allows better control over the size of the unfixed neighbourbood N. An unpropagated α might have many implied literals not in α, and each such literal reduces the possible number of feasible solutions in N since these literals are implicitly fixed in N.

We start with a fixed set α of size about 80% of the total number of variables in the problem, i.e., with $\delta = 0.8$. After ten consecutive rounds of LNS where no new $best$ solution is found, we reduce α's size by reducing δ by 0.1. Systematically changing neighbourhoods is a well-known metaheuristic in local search algorithms called variable neighbourhood search [128]. We decided to start with a small neighbourhood (a large fixed set) because small neighbourhoods will usually be easier to solve with C_MaxSlv. Notice that once δ gets to 0.1 and we encounter ten consecutive rounds with no new best solution, δ will be set to 0 and the entire problem will be solved by C_MaxSlv optimally. In our experiments this very rarely happened (less than 0.5% of all instances). We also found that dynamically decreasing δ worked considerably better than using a fixed value. For clarity, our final neighbourhood selection policy is given in Algorithm 3.

One other minor improvement not shown in Algorithm 3 is that the selected fixed set α generally falsifies some soft clauses (any soft clause c where $\forall l, l \in c \rightarrow \neg l \in \alpha$). Let $cost(\alpha)$ denote the sum of the costs of the clauses falsified by α. (Note that since α is a partial assignment some clauses are neither satisfied nor falsified by α). This means in any round of LNS in Algorithm 2 the cost of the best solution found in each neighbourhood is at least $cost(\alpha)$, i.e., $cost(\alpha \cup N\_best) \geq cost(\alpha)$. Hence, if $cost(\alpha) \geq cost(best)$ there is no point in searching the neighbourhood defined by α—that neighbourhood cannot yield a better solution. Hence, we check $cost(\alpha)$ and immediately fail that round without calling $C\_MaxSlv(F|_\alpha)$ if $cost(\alpha) \geq cost(best)$. It is also possible to reject α when its cost is too close to $cost(best)$, but exploring this option is left for future work.

Algorithm 3: Neighbourhood selection policy

```
1
2  Input: weighted CNF F, current feasible solution π, δ desired size of fixed set
3  α = ∅                                          /* initialize fixset to empty */
4  for each l ∈ π: l^w = 0
5  for each c ∈ soft(F) do
6      if π ⊨ c then
7          for each l ∈ c: l^w = l^w + c_w
8      else
9          for each l ∈ c: (¬l)^w = (¬l)^w − c_w
10         /* Note that if π ⊭ c then l ∈ c → ¬l ∈ π */
11 for each l ∈ π: prob(l) = 2^{l^w} / Σ_{l∈π} 2^{l^w}
12 hards = hard(F)
13 while size(α) < δ|π| do
14     select l ∈ π randomly with prob(l).
15     hards = UP(hards ∪ (l))                    /* add l to hards and unit prop*/
16     α = all units in hards
17     /* Note the units of hards includes all selected literals*/
18     /* and all of their unit implications*/
19 return α                                       /* Final selected fixed set */
```

5.3.4 Other Possible Neighbourhood Selection Policies

It would perhaps be more accurate to keep a list of soft clauses and remove those that are satisfied by each literal as that literal is added into the fixed set, i.e., after adding a literal l to the fixed set, we would re-compute the preference weighting for all remaining literals using the updated soft clauses list. However, this is more costly requiring multiple passes of the soft clauses during the neighbourhood selection process, whereas our neighbourhood selection process requires just one pass. In our experiments, doing only one pass of the soft clauses seemed to work better so we decided to proceed with that.

An alternative approach of neighbourhood selection would be to select some subset of the soft clauses satisfied in the initial solution to be a fixed set. These soft clauses would then be treated as hard clauses (i.e., we would force these soft clauses to be satisfied), and C_MaxSlv would find an optimal way to satisfy the remaining unfixed soft clauses. We experimented with this and found it was not quite as good as fixing literals, so we decided to focus on literal fixing for the rest of our implementations.

5.3.5 Incremental MaxSAT

There is increasing interest and an increasing body of literature around the idea of incremental MaxSAT [129, 17]. The LNS for MaxSAT algorithm we have proposed is an application of incremental MaxSAT because we are calling C_MaxSlv iteratively on instances which are reduced versions of the formula F. These formulas will often be the very similar because our neighbourhood selection policy involves an exponentially weighted random distribution, where the weighting corresponds to how much a literal contributes to satisfying clauses minus how much it contributes to falsifying clauses. Because the solutions will often be similar in the same neighbourhood, the weighting on the literals will be very similar and thus the randomly selected fixed set will be similar.

Making the fixed set of literals "assumptions" at the MaxSAT level could result in the sharing of learnt clauses between MaxSAT invocations when using an incremental MaxSAT solver, which currently we are not taking advantage of. Additionally, one can reuse lower bound and upper bound information in between invocations and can also share the cores from previous MaxSAT iterations that have all of their clauses

(represented by relaxation literals) assumed in the current MaxSAT invocation. This leaves a lot of potential for improvement of the LNS for MaxSAT algorithm.

5.4 Relationship to Branch-and-Bound MaxSAT

There have been some interesting developments recently in branch-and-bound MaxSAT algorithms that have allowed it to combine with core-guided or implicit hitting set solvers to improve the state-of-the-art [130]. Our LNS for MaxSAT algorithm shares some similarities with the branch and bound algorithm.

In the branch and bound algorithm, one branches on variables in a search and uses an approximation (such as unit propagation) to check if the lower bound under that variable branching is worse than the best upper bound found so far. If so, you can stop and prune everything in the search tree under that variable branching, since the optimal solution can not be there.

In our LNS for MaxSAT algorithm, one can look at the fixed set as a variable branching, and we are conducting a MaxSAT search under that variable branching. Instead of doing a complete MaxSAT search, one could borrow elements from the branch and bound search and try to quickly determine a lower bound under the fixed set "variable branching". If that lower bound is worse than the best upper bound found so far, then one can prune all solutions with that fixed set from the search space. Borrowing this and other elements from the branch and bound algorithm has potential to improve LNS for MaxSAT further, but we have left that to future projects.

5.5 Experiments and Results

In this section, we will first talk about our setup and benchmarks that we used for each of the experiments. The first set of experiments in Section 5.5.2 tests whether or not LNS is able to improve individual MaxSAT solutions faster than other state-of-the-art anytime solvers can, and furthermore how it improves the solution based on how high quality the solution is. Using the positive results of the first experiment, the next set of experiments in Section 5.5.3 aims to find the optimal time budget for LNS assuming no overhead in a 60 second or 300 second context. The last experiment in Section 5.5.4 actually builds a solver and budgets the amount of time determined from the previous experiment in the 300 second context. In spite of having some overhead, this last experiment showed we are still able to improve all three state-of-the-art anytime solvers, and it is the main result of this chapter.

5.5.1 Setup and Benchmarks

All of our experiments were run on 2.7GHz Intel cores with 300 second CPU time and 16GB memory limits. The MaxSAT evaluations also test a time out of 60 seconds, and we experimented with that time out as well and showed some of our data in Figure 5.4. However our approach involves many calls to C_MaxSlv in the main loop of LNS and in our current implementation these calls are not optimized for incrementality. In particular, each call to C_MaxSlv requires processing the input instance from scratch and re-initializing the solver with a new reduced instance. For the many large instances appearing in the evaluation this imposed an overhead that significantly impacted the shorter 60 second runs. So much so that we did not find LNS to be effective in the 60 second context. There are a number of ways a more sophisticated implementation could reduce this overhead, so we left making LNS effective in this shorter time frame as future work. In

general, properly addressing this issue requires improved techniques for incremental MaxSAT solving which is an important area for future research.

The benchmarks we used for the first two experiments were the 253 weighted and 262 unweighted instances from the Incomplete Tracks of the 2020 MaxSAT Evaluation [123] (we used the 2020 benchmarks instead of 2021 because there are more total instances in the 2020 benchmarks). In order to avoid overfitting, we did all tuning of our neighbourhood selection policy on the instances from the 2018 and 2019 MaxSAT Evaluations before settling on the policy described in Section 5.3.2. We implemented our LNS algorithm as described in Section 5.3 using MaxHS [122] as the complete solver C_MaxSlv.

Computing Scores

To measure how effective our approach was at finding good solutions, we used the same scoring system as used in the MaxSAT evaluations. In particular, the score a solver S achieves on an instance i is $score(S, i) = \frac{bk_i+1}{sc_i+1}$, where sc_i is the cost of the best solution found by S for instance i and bk_i is the cost of the best known solution for instance i. The values bk_i were supplied by the MaxSAT evaluation. On an instance i a score of 1 indicates that S found a solution of cost equal to the best known, and a score of 0 indicates that S was unable to find any feasible solution (in this case $sc_i = \infty$). The score of S on a set of instances I is simply its average score over these instances: $score(S, I) = \frac{\sum_{i \in I} score(S,i)}{|I|}$.

5.5.2 Analyzing LNS' Improvement on Solutions

Our first experiment is designed to evaluate how effective LNS is at improving feasible solutions, and to compare this to how effective a representative state-of-the-art anytime solver is at the same task. In particular, when an anytime solver AT_MaxSlv is run on an instance it will emit a sequence of monotonically improving solutions (decreasing solution costs) over its 300s run. If we take a solution π emitted at time t (measured in seconds) we can use this sequence to determine the best solution π' emitted at time $t + \Delta$ for any interval Δ. (π' might be the same as π if no better solution has been found Δ seconds after π was found). We can then measure the improvement in solution cost achieved from solution π, $cost(\pi) - cost(\pi')$, as a function of time Δ. We can do the same for the LNS solver by giving it π as its initial solution.

We chose TT-Open-WBO [131], one of the best incomplete solvers in recent MaxSAT evaluations, as the representative state-of-the-art anytime solver to compare LNS "rate of improvement" with.[4]

We ran TT-Open-WBO on each instance from the 2020 MaxSAT incomplete solver evaluation for 600s, combining the weighted and unweighted instances to obtain 515 instances.[5] For each instance we stored the timestamped sequence of solutions TT-Open-WBO emitted. This allowed us to use any solution emitted by TT-Open-WBO within the first 300s and still have a full 300s "future" for that solution. We took all solutions-instances pairs where the solution was emitted by TT-Open-WBO in the first 300 seconds and further filtered them to control their total number. In particular, if multiple solutions for that instance were found in the same 5-second interval, only the best one was kept. We then ran our LNS algorithm for 300s on each of these solution-instance pairs. We similarly recorded the sequence of solutions emitted by LNS for each solution-instance pair over its 300s run. This gave us a set of solution-instance pairs along with a 300s future evolution of those solutions in TT-Open-WBO and in LNS.

To compare the rate of improvement of TT-Open-WBO with that of LNS, we split the solutions into four buckets based on solution quality. The reason for separating the solutions is that poor solutions typically can

[4] Other incomplete solvers gave similar results.

[5] For this experiment we were not interested in distinguishing these two cases.

be improved at a much faster rate by either solver than good solutions. The first bucket had solutions with a score between 0.0 and 0.25, the second with scores between 0.25 and 0.5, the third with scores between 0.5 and 0.75, and the fourth with scores between 0.75 and 1.0. Then, for each bucket of solutions, we kept only one randomly picked solution per instance, since some instances contributed multiple solutions to the same bucket which would bias the results towards those instances. This left the first bucket with 75 solutions, the second bucket with 109 solutions, the third bucket with 136 solutions, and the fourth bucket with 453 solutions. We plot in Figure 5.2 the average score obtained over all solution-instance pairs in the bucket and how it increased with time as LNS worked on them, compared to the average score that TT-Open-WBO achieved over time for the same solution-instance pairs.

We can observe that LNS is able to achieve a good improvement of the solutions in all buckets. In fact, for the first 5-10 seconds it is able to improve the solution quality faster (as fast for the 0.5–0.75 solutions) than TT-Open-WBO. However, for the lower quality solutions (those with scores < 0.75), its rate of improvement quickly diminishes and its final level of improvement was considerably worse than TT-Open-WBO. This makes intuitive sense. LNS intensely searches neighbourhoods close to the solution it starts with, and it can take it a long time to move far away from that solution. TT-Open-WBO on the other hand has much more mobility in the search space and is able to move away from bad initial solutions more quickly.

When we examine the highest quality solutions, those in the 0.75–1 bucket, we see that TT-Open-WBO can achieve some improvement to these solution. But its rate of improvement is no better than LNS. More importantly, its final level of improvement is worse than LNS. We conjecture, that for these higher quality solutions there are not as many better solutions, and it is not as easy for TT-Open-WBO to find them. Whereas LNS, by searching more systematically, is more effective at finding these more sparse better solutions.

This data suggests that a promising way to use LNS is to ensure that the anytime solver has enough time to produce a higher quality solution and then use LNS to improve that solution. In our next experiment we tried giving the anytime solver various fixed amounts of time (i.e., running it for the same amount of time on each instance) before finishing up with using our LNS solver. An alternative that would be worth investigating in future work would be to try to detect when the anytime solver is plateauing in its ability to improve the solution and invoke the LNS solver at that point.

5.5.3 Finding the Optimal Amount of Time to Budget for LNS

Our second experiment used the same data as experiment one. As mentioned above it involved running TT-Open-WBO for some amount of time, before switching to our LNS solver for the remaining amount of time and finally outputing the best solution found. In Figure 5.3, we show the total score achieved when taking the best solution for each instance emitted by TT-Open-WBO within $300 - t$ seconds and then allowing our LNS solver t seconds to improve those solutions (for different values of t). Note, that when $t = 0$ we obtain the baseline performance of TT-Open-WBO, and this baseline is indicated in Figure 5.3 by the horizontal dashed line. Figure 5.3 clearly shows that stopping TT-Open-WBO early and running LNS for some amount of time resulted in a performance gain up until a threshold of about 50 seconds after which the performance starts to decline.

Since the LNS solver is based on MaxHS, which uses a different MaxSAT algorithm [34], some of the performance gain obtained by switching from TT-Open-WBO to LNS could be simply because we are using two different solvers and taking the best solution computed by either of them. This is known as the **portfolio effect**. To demonstrate that the performance gain is not solely from the portfolio effect, we also show in Figure 5.3 the score achieved by running TT-Open-WBO for $300 - t$ seconds and then running plain MaxHS

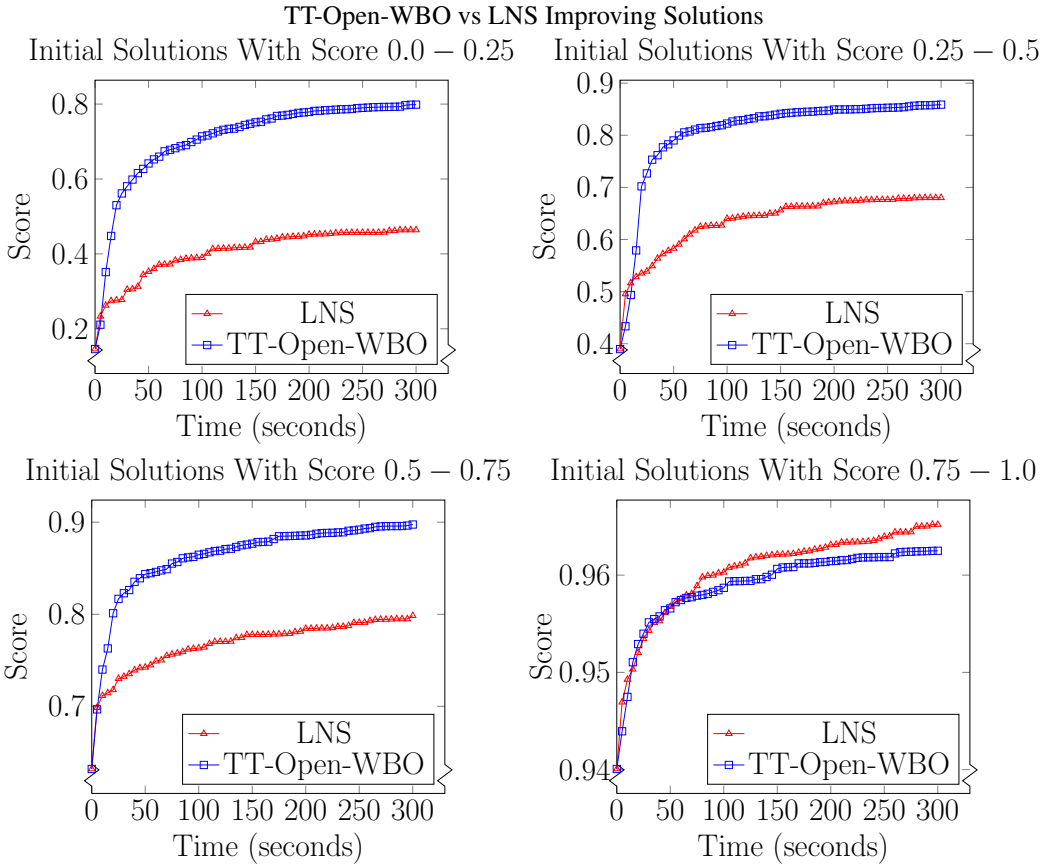

Figure 5.2: Comparison of TT-Open-WBO continuing to try to improve its solution vs our LNS solver trying to improve the same solutions, separated into four buckets of solution quality. The top left grid is the bucket of lowest quality initial solutions, the top right is the bucket of second lowest quality initial solutions, the bottom left is the bucket of second highest quality initial solutions, and the bottom right is the bucket of highest quality initial solutions. As the solution quality increases, the rate at which LNS improves solutions becomes more and more favourable compared to the rate at which TT-Open-WBO improves solutions. Only in the highest quality bucket is it able to improve solutions at a faster rate than the base solver, TT-Open-WBO.

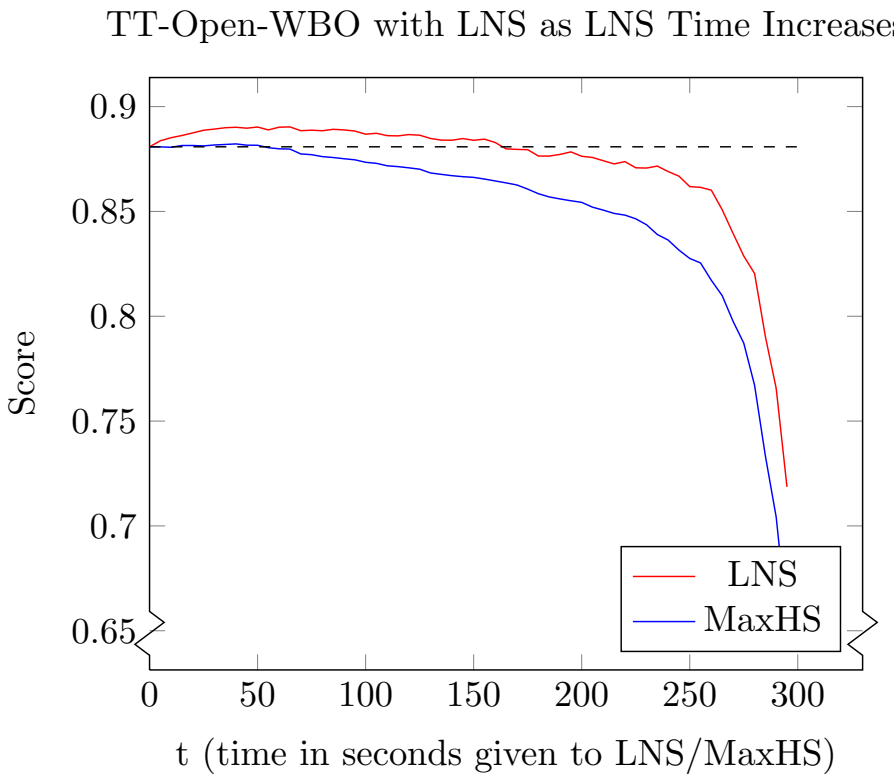

Figure 5.3: Total score achieved running TT-Open-WBO for $300 - t$ seconds and then running LNS for t seconds on its best solution. We also show the results of running TT-Open-WBO for $300 - t$ seconds and then running plain MaxHS for t seconds.

for t seconds. Note that MaxHS also outputs intermediate solutions, so in cases where MaxHS can not solve optimally, we take the best solution it output and use that as its solution. We can see that using LNS dominates using MaxHS for all values of t, indicating the performance gain is not solely due to a portfolio effect.

We also show similar data for the 60 second timeout instead of the 300 second timeout in Figure 5.4. The graph shows a very similar shape to the 300 second timeout data and shows us that the optimal time to let LNS run in the 60 second timeout would be somewhere between 15 seconds. Here, the evidence that our technique is not gaining due to a portfolio effect is even more evident, as the graph shows that no amount of running MaxHS is beneficial in the 60 second timeout setting to improve the anytime score.

However, as stated earlier, because of the overhead involved in stopping one solver and calling the other, our LNS in its current implementation does not work as well in the 60 second timeout setting and thus, we leave it to future work to fix this and do further experiments to build an optimal solver for the 60 second timeout setting.

5.5.4 Building Solver With LNS and Comparing With the State-of-the-Art

For the last set of experiments, we try to improve the performance of three of the best performing anytime MaxSAT solvers from the 2020 and 2021 MaxSAT evaluations: TT-Open-WBO, Loandra [132], and satlike [133]. Loandra is a SAT-solver based MaxSAT solver, while TT-Open-WBO and satlike use a combination of SAT-solving and local-search. We will describe each of these solvers in a little more detail below.

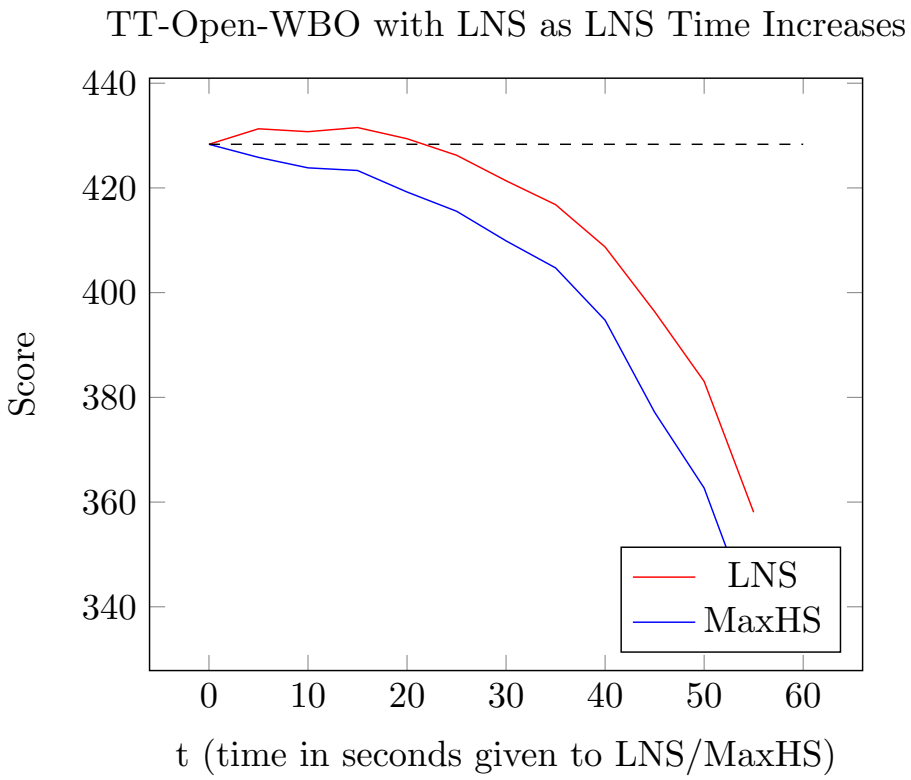

Figure 5.4: Total score achieved running TT-Open-WBO for $60 - t$ seconds and then running LNS for t seconds on its best solution. We also show the results of running TT-Open-WBO for $60 - t$ seconds and then running plain MaxHS for t seconds.

Loandra. Loandra combines the core-guided complete MaxSAT algorithm with the linear-sat-unsat search algorithm incomplete MaxSAT algorithm. In the version we used, Loandra also first sends the formula to a MaxSAT preprocessor, maxpre [134] which simplifies the MaxSAT instance. Next, is the main algorithm of Loandra. In the first phase of the main algorithm, Loandra will call a core-guided solver to transform for the formula into something that is hopefully easier to solve. The transformed formula after collecting a bunch of cores has implicitly closed the gap between the lower bound and the upper bound. This transformed formula is then given to linear-sat-unsat search, which will continually add cardinality constraints to find a better and better upper bound solution until it reaches an unsatisfiable state (meaning it can not improve the upper bound any further).

Satlike. Satlike combines a stochastic local search incomplete MaxSAT algorithm with linear-sat-unsat search incomplete MaxSAT algorithm. First, satlike will call a regular SAT solver to get a feasible solution. It will then take this solution and feed it into a stochastic local search algorithm developed for MaxSAT in order to try to improve that solution. After running the first phase for some amount of time, it will then take the best feasible solution found and feed it into the second phase, which runs a linear-sat-unsat search algorithm. The best solution found during this phase is then outputted.

TT-Open-WBO-inc. TT-Open-WBO-inc is based on linear-sat-unsat search algorithm, with many heuristics added over a number of years to improve performance. In the version we used, it also uses the stochastic local search algorithm from satlike in order to improve its initial feasible solution for 15 seconds, before switching to the linear-sat-unsat search algorithm. The main algorithm also uses concepts from polosat [135] to get further improvements.

We implemented three solvers, LNS-TT-Open-WBO[6], LNS-satlike, and LNS-Loandra, which ran the corresponding anytime MaxSAT solver for 250 seconds and then used our LNS solver to improve the best solution they found for the final 50 seconds. The 50 second limit for LNS was based on the data shown in Figure 5.3. We also ran the base solvers TT-Open-WBO, satlike, and Loandra for 300 seconds each in order to compare them to the LNS hybrid versions. For this last set of experiments, we used the union of all weighted and unweighted instances from the 2020 and 2021 MaxSAT evaluations. After removing duplicates, this left 352 weighted instances and 374 unweighted instances for a total of 726 instances. The results are given in Table 5.1. For completeness, we also show the results separated into 2020 and 2021 in Tables 5.2 and 5.3

For both the weighted and unweighted instances, the total scores obtained by the LNS versions were an improvement over the baseline for all three solvers. Although the improvements may appear small, even modest improvements are difficult to achieve on MaxSAT competition benchmarks. To give some context, in the four tracks of the 2020 MaxSAT Evaluation, the average difference between the winning solver and the runner-up was 0.0075 in average score, and in 2021 it was 0.0178. The LNS solvers are also finding better solutions than any other solver had previously found. In total, the three LNS solvers were able to improve the best known solution for 55 of the 515 total instances from 2020 and 40 of the 306 total instances from 2021. On average in these cases the best known solution was improved by 3.7% for 2020 instances and 4.8% for 2021 instances; the maximum improvement we saw to a previously best known solution was 85.7%. These results make it clear that budgeting some time for LNS to run on the best solution obtained from any of these anytime solvers is a good strategy to improve their performance (on these benchmark instances). Since these three anytime solvers were the best performing solvers in the evaluation, we can see that our approach improves the state-of-the-art.

[6]Source code can be found at `https://github.com/rgh000/MaxSAT_LNS`.

Solver	Weighted	Unweighted	Total
TT-Open-WBO	.855	.853	.854
LNS-TT-Open-WBO	**.866**	**.864**	**.865**
satlike	.836	.850	.843
LNS-satlike	**.855**	**.867**	**.861**
Loandra	.835	.825	.830
LNS-Loandra	**.848**	**.845**	**.846**

Table 5.1: Results for the union of the 2020 and 2021 MaxSAT Evaluation instances for TT-Open-WBO, satlike, and Loandra along with the LNS versions of each. The LNS versions' scores (in bold) are an improvement for both weighted and unweighted instances for each solver.

Solver	Weighted	Unweighted	Total
TT-Open-WBO	.886	.877	.881
LNS-TT-Open-WBO	**.897**	**.892**	**.894**
satlike	.873	.869	.871
LNS-satlike	**.894**	**.888**	**.891**
Loandra	.841	.836	.838
LNS-Loandra	**.852**	**.862**	**.857**

Table 5.2: Results for 2020 MaxSAT Evaluation instances for TT-Open-WBO, satlike, and Loandra along with the LNS versions of each. The LNS versions' scores (in bold) are an improvement for both weighted and unweighted instances for each solver.

Solver	Weighted	Unweighted	Total
TT-Open-WBO	.810	.836	.823
LNS-TT-Open-WBO	**.819**	**.838**	**.829**
satlike	.778	.839	.809
LNS-satlike	**.794**	**.850**	**.822**
Loandra	.814	.813	.814
LNS-Loandra	**.827**	**.827**	**.827**

Table 5.3: Results for 2021 MaxSAT Evaluation instances for TT-Open-WBO, satlike, and Loandra along with the LNS versions of each. The LNS versions' scores (in bold) are an improvement for both weighted and unweighted instances for each solver.

5.6 Related Work

In a paper by Demirović et al. [115], an LNS method that uses a MaxSAT solver as part of its workflow was used to improve initial solutions for high school timetabling problems, but not used to solve MaxSAT problems directly. In finding initial solutions to these high school timetabling problems, domain-specific information was taken advantage of rather than converting the whole problem directly into MaxSAT from the start.

In a paper by Ramaswamy et al. [136], a MaxSAT solver is used to make local improvements to treewidth decompositions. Their work is similar to using LNS except that the neighbourhood is defined by the structure of the graph they are working with.

In a paper by Björdal et al. [137], it was shown how LNS can be used to help solve very difficult satisfiability problems, but LNS was not used to solve MaxSAT problems. It would be interesting to see if some of the ideas that were successful in solving very difficult satisfiability problems with LNS could be adapted to solve MaxSAT problems.

In none of these works, however, do we find a direct application of LNS to MaxSAT solving.

5.7 Discussion

In this chapter, we have hypothesized that anytime MaxSAT solvers often stay in the same local region of the search space as they find new solutions. This leads to a lot of repetition in the computations of the solver as it tends to find very similar solutions. Because of this, we have shown we can reduce the formula down and more exhaustively search a local region of the MaxSAT solution space with an algorithm based on large neighbourhood search. This leads to the most important contribution of the chapter, which is introducing the first LNS-based algorithm for anytime MaxSAT and exploring its effectiveness, which fills a gap in the literature.

We have demonstrated experimentally that running LNS for a small amount of time can sometimes improve suboptimal MaxSAT solutions faster than those solvers can improve their own respective solutions. More specifically, we showed that high quality solutions can be improved by LNS better than any of the other anytime algorithms can improve them, but that lower quality solutions are not improved as well by LNS. This is important, as it means LNS is a good addition to other anytime algorithms, and that budgeting some time for LNS is a good idea. By implementing our LNS algorithm to operate on the best solution from the anytime solver, we were able to improve the anytime performance of three of the best performing anytime solvers, Loandra, satlike, and TT-Open-WBO, on a wide range of both weighted and unweighted benchmarks from the MaxSAT Competitions.

Furthermore, we found new solutions that no other anytime solver was able to find, further demonstrating that the benefits of LNS are not captured by any of the other techniques. Because high quality solutions are often achieved very quickly by good incomplete solvers, but then those solvers get stuck in local maxima, it makes sense that switching to another technique like LNS once a high quality solution is achieved would help find new solutions that are unreachable by the other incomplete algorithms.

5.7.1 Future Work

The approach we have described uses LNS once to improve a feasible solution produced by another anytime solver. However, with more implementation work a tighter integration between the solvers could be achieved.

In particular, our results indicate that LNS can often more rapidly improve even a poor solution in the first 5 to 10 seconds. Potentially this could be exploited in an anytime solver, by having it invoke LNS for a brief time to improve each solution it finds. This idea has even more potential if ways could be found for the anytime solver to take advantage of the LNS improved solution, e.g., by adjusting its search for the next better solution. It is an interesting research project to find effective ways of accomplishing this.

Aside from using multiple initial solutions, one could also try multiple different solvers for C_MaxSlv. We used MaxHS because it is one of the best performing complete MaxSAT solvers and also outputs some anytime solutions periodically as it runs, but other complete solvers still might work better and potentially even an anytime solver could be used instead (although you would lose the guarantee that you have found the best solution in the neighbourhood of solutions, you could still try the anytime solver to see if it can find a "good" solution in a short amount of time in the neighbourhood being examined). Allowing the assignments to drift away from the solution space similar to how stochastic local search does is also possible, but would require further experiments to determine if it is beneficial or not.

Another area for future work is further development of incremental MaxSAT solving. The LNS solver is called many times and building an incremental framework to allow reusing the work of previous calls to C_MaxSlv (e.g., previously discovered cores or learnt clauses) could result in a further performance improvement. For example, one could try to learn clauses after each returning MaxSAT call from C_MaxSlv, since we know that a clause containing the negation of all literals in the fixed set is automatically not going to lead to a better solution. Some of the new developments in learning clauses during branch-and-bound MaxSAT would be interesting to explore for our LNS algorithm as well.

Finally, it is known that LNS can get stuck exploring a local part of the solution space when the neighbourhood selection policy does not allow for enough diversity. We observe from our experiments that this is probably what is happening to our neighbourhood selection policy and one way to combat this might be to use some strategies from the literature that have worked in other contexts, such as adaptive large neighbourhood search [138] or using predictive machine learning models to select better neighbourhoods.

Chapter 6

Conclusion

6.1 Summary

In this dissertation, we explored different types of redundant work in SAT solvers and other related propositional logic solvers. We not only identified redundant work that was being done in multiple different types of solvers, but we then introduced new techniques that helped reduce the redundant work we had identified, and ended up improving the performance of the propositional logic solvers empirically on diverse benchmark sets. These solvers included SAT solvers, MUS extractors, complete MaxSAT solvers, and anytime MaxSAT solvers.

In some cases, the techniques we introduced to reduce the redundant work did so by preventing repeated computations from occurring. An example of this was in assumption-based SAT in Chapter 3, where we noticed that while unit propagating the assumptions, each clause on each assumption's watchlist gets accessed numerous times before the solver realizes the clause is unit, so we enqueued all assumptions at once to avoid having to re-visit clauses on watchlists repeatedly. Another example of this was in anytime MaxSAT in Chapter 5, where we noticed that because anytime MaxSAT solvers are usually staying in local regions of the solution space, they are finding very similar solutions over and over again. To address this, we introduced a technique to reduce the formula down and more exhaustively search those local regions of the MaxSAT solution space with an algorithm based on large neighbourhood search.

In other cases, we reduced redundant work by reusing previous computations, such as in trail saving in Chapter 4 (and assumptions-level trail saving in Chapter 3), where we noticed that the same unit propagation steps were being reproduced by SAT solvers over and over again, so we stored and reused part of the assignment trail to eliminate a lot of redundant work. We also found other ways to re-use this saved trail, such as to "lookahead" and predict which branches in the search tree have conflicts.

In each case, all of our techniques were able to improve multiple state-of-the-art solvers in their respective domains. For the assumption-based SAT techniques in Chapter 3, we were able to make a significant improvement in two state-of-the-art complete MaxSAT solvers, MaxHS and RC2, on a wide set of weighted and unweighted benchmarks from the MaxSAT competitions. For the trail saving techniques introduced in Chapter 4, we were able to make a fair improvement in two state-of-the-art SAT solvers, cadical and MapleSAT, on a wide set of benchmarks from the SAT competitions. For the LNS algorithm we introduced for anytime MaxSAT in Chapter 5, we were able to improve three state-of-the-art anytime MaxSAT solvers, Loandra, TT-Open-WBO-inc, and satlike, on a wide range of weighted and unweighted benchmarks from the

MaxSAT competitions.

Aside from introducing and implementing the aforementioned techniques, we also had to prove the soundness of the techniques and, in some cases, prove that they held certain desirable properties. The key to proving soundness of enqueueing all assumptions at once was the fact that it lead to the exact same set of unit propagants as enqueuing assumptions one by one (i.e., the standard approach). The soundness of our trail saving technique relied on the fact that any portion of a previous saved trail can still be re-used as long as its literals have not been falsified. Aside from soundness, we also showed that trail saving holds certain invariants that were traditionally held in CDCL SAT solvers, unlike a new technique, chronological backtracking, which does not. The soundness of our large neighbourhood search algorithm for MaxSAT was based on the fact that fixing a certain number of variables in a MaxSAT formula leaves a subproblem (which is also a MaxSAT formula) that, when combined with the variables that were fixed, will be a solution to the original problem.

6.2 Impact

The techniques introduced in this thesis have been used or built upon by independent groups since their publication. UWrMaxSat [22], the winner of the 2020 weighted track of the MaxSAT evaluation, used the assumptions techniques from Chapter 3. The trail saving ideas from Chapter 4 have been extended to SMT solving [23]. The ideas from Chapter 3 and Chapter 4 also seems to have help inspire ideas used in Incremental Lazy Backtracking [24].

Although many of our results might seem like small improvements, it is small incremental improvements accumulating over time that have yielded enormous improvements over the last 10 to 15 years. For example, in results shown at the 2018 MaxSAT Evaluation [139], the best MaxSAT solvers from 2017 were able to solve about three times more instances than the best MaxSAT solvers from 2008 on the same set of benchmarks, and this trend has continued since then.

6.2.1 Contributions

The most significant contributions have been highlighted in each chapter, but we will summarize what we believe are especially important contributions for each chapter below.

For Chapter 3: Speeding Up Assumption-Based SAT

- we identified an inefficiency in the way assumptions are unit propagated, i.e. they repeat the same unit propagation steps when the solver backtracks into the assumptions levels
- we addressed this inefficiency by introducing a new technique to enqueue all the assumption literals at once
- we also introduced a way to save part of the trail that has the assumptions levels and reuse it in future iterations to save redundant unit propagation steps
- experimentally, we were able to improve the performance of two state-of-the-art MaxSAT solvers, RC2 and MaxHS, by a significant amount on a wide range of benchmark instances coming from the MaxSAT Competitions

- we determined that our technique to enqueue all assumptions at once produces smaller cores on average than the standard approach, which is a surprising result and probably also helps improve the performance

For Chapter 4: Trail Saving on Backtrack

- we identified repeated work that occurs during the construction of the assignment trail in CDCL SAT solvers
- we showed how chronological backtracking addresses this redundant work but also has some undesirable effects on search
- we introduced a new technique, trail saving, that reuses parts of an assignment trail and addresses the same redundant work that chronological backtracking does, but without some of the potentially negative side effects
- we introduced enhancements to trail saving which improved its performance even more, including saving the trail over multiple backtracks, accounting for poor reason quality saved on the trail, and using the trail to perform lookaheads
- experimentally, we managed to improve two state-of-the-art SAT solvers, cadical and MapleSAT, on a wide range of benchmarks from the SAT competitions
- we did further analysis to show that the performance improvement realized by trail saving was really a speedup in unit propagation, as hypothesized

For Chapter 5: LNS for MaxSAT

- we identified redundant work being performed in anytime MaxSAT solvers by showing they tend to stay in the same local region of the solution space and produce multiple solutions which are extremely similar. We combat this by exhaustively searching a local region of the search space with an application of LNS to MaxSAT, the first LNS-based algorithm for MaxSAT
- we developed a neighbourhood selection policy that works much better than a random policy (and also other alternative policies that we tried)
- we showed that our LNS for MaxSAT improves higher quality solutions faster than other anytime algorithms can improve those same solutions, but also that LNS for MaxSAT improves lower quality solutions slower than other anytime algorithms can
- we were able to improve three state-of-the-art anytime MaxSAT solvers, TT-Open-WBO, SatLike, and Loandra, on a wide range of benchmarks from the MaxSAT Competitions by budgeting some time for our LNS algorithm to run

Within each of the technical chapters, we have discussed some of the potential future work. We will summarize what we think are the most promising of these below.

For Chapter 3: Speeding Up Assumption-Based SAT

- exploring how to further minimize final conflict clauses (and thus minimize cores), perhaps by using clause minimization or other resolution-based techniques, is an interesting avenue and could lead to significant performance improvements
- trying our technique to enqueue all assumptions at once in other incremental SAT domains, such as bounded model checking
- further exploring why our technique (enqueueing all assumptions at once) is not as effective at improving the solver when abstract cores are used, and finding a remedy for it, is an important question left unanswered

For Chapter 4: Trail Saving on Backtrack

- using the saved trail to make inprocessing techniques faster could make saving the trail even more useful for SAT solvers
- extending trail saving to other domains such as SMT solving, CSP solving, or model counting is worth exploring
- learning multiple different clauses from the trail, especially if one keeps multiple trails, is possible when there are multiple different contradicting literals on the old trail(s)

For Chapter 5: LNS for MaxSAT

- using multiple different solutions as initial solutions for the LNS rounds to operate on could be beneficial. One would then have to do further analysis to find the optimal amount of time to run each round before going back to the other solver to get another initial solution
- further development of incremental MaxSAT to save work in between MaxSAT invocations (also potentially extending ideas of trail saving from Chapter 4 to incremental MaxSAT)
- using predictive machine learning models to select better neighbourhoods than the simple policy we currently have could get to better parts of the solution space faster

6.3 Future Research Directions

Taking Redundant Work Further

There are other areas of redundancy that can be found in propositional logic solvers that are highly related to some of the work done in this thesis and should be explored. One idea that shows promise is trying to to extract multiple cores at once, which will allow one to save time between calls in the LNS for MaxSAT algorithm. Often times a lot of the exact same unit propagation steps are performed while extracting consecutive cores, which is redundant work. Aside from unit propagation steps, often times the conflict graphs between consecutive SAT calls share a lot of similarities, and so finding a way to use an extracted proof [140] from one SAT call to more quickly find the next core (or a collection of cores) is a very interesting project worth exploring.

In a way that is analogous to some of the work we have presented in this thesis, one could try to reuse work done in between the MaxSAT calls in LNS for MaxSAT algorithm. We have already shown that the LNS for MaxSAT algorithm is an instance of incremental MaxSAT, and taking that one step further one could

try to prevent fixed sets from being reused (or encourage a better diversity of fixed sets), reuse cores, reuse lower and upper bounds, save the trail from propagating other fixed sets and use it to guide the selecting of new fixed sets, etc. There are numerous opportunities for reducing the redundant work that happens between MaxSAT calls in the published version of the algorithm, and this is another area that could be fruitful to pursue.

Redundant Work in Other Domains

Redundant work will inevitably appear in other related areas of artificial intelligence beyond propositional reasoning, and perhaps even in other areas of computer science. We close with some thoughts about how others can use some of these ideas in those other domains.

In order to recognize where some of the redundancies are happening, a good place to start is to examine the execution of the state-of-the-art solvers and identify computations which are being repeated over and over. For example, when we examine the execution of a SAT solver performing its main search, we saw that the same sequence of decisions (and unit propagants) were appearing on the trail over and over again, which would mean exactly the same underlying work to produce those was being repeated. Before attempting to extend something like trail saving to CSP solving or model counting, one might want to examine the execution of the solver and check if there is a lot of repetition in the assignments being built up (and also repetition in the way they are propagated).

The next question one must address before undertaking a project to reduce this redundancy is to measure how much potential impact removing this redundancy can have on the performance of the solver, and determine whether or not this improvement is significant. For example, in many machine learning contexts, removing some redundancy that results in a polynomial speedup and will take the training time from 2 weeks to 1 week is unlikely to be significant (after all, we can just wait longer or increase the computational power). However, in the context of anytime MaxSAT solving, speeding up can mean the difference between solving certain instances in time or not, and many applications of anytime MaxSAT rely on the solver producing solutions very quickly, even where there are limited resources.

Lastly, finding a way to address the redundancy often requires creativity and understanding of how the solver works in the specific domain. For example, in our setting, we understood that repetitive propagation steps as a result of assumptions could be removed by giving the unit propagation process access to the information of all assumptions that are about to be made ahead of time. Similarly, one would have to examine if such information could help speed up other types of propagation that occurs in other solvers, such as CSP solvers.

Concluding Statement

Avoiding redundant work in solvers and attempting to reuse the work that has already been computed by solvers remains a fruitful area for future research. Whether it is brand new ideas or the extension of the ideas we present in this thesis, we believe that exploiting that redundancy will play a significant role in continuing the rapid improvement of propositional logic solvers. We hope that the findings in this thesis are encouraging to the propositional reasoning community and beyond to continue to look for patterns, redundancy, and repetitive work, so that we can continue to move forward the frontier of what is possible in computer science.

Bibliography

[1] Koen Claessen et al. “SAT-solving in practice”. In: *2008 9th International Workshop on Discrete Event Systems*. IEEE. 2008, pp. 61–67.

[2] Armin Biere and Daniel Kröning. “SAT-based model checking”. In: *Handbook of Model Checking*. Springer, 2018, pp. 277–303.

[3] Nina Narodytska et al. “Verifying properties of binarized deep neural networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[4] Marijn JH Heule, Oliver Kullmann, and Victor W Marek. “Solving Very Hard Problems: Cube-and-Conquer, a Hybrid SAT Solving Method.” In: *IJCAI*. Vol. 17. 2017, pp. 228–245.

[5] Armin Biere et al. “Bounded model checking.” In: *Handbook of satisfiability* 185.99 (2009), pp. 457–481.

[6] Carla P Gomes, Ashish Sabharwal, and Bart Selman. “Model counting”. In: *Handbook of satisfiability*. IOS press, 2021, pp. 993–1014.

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

[8] Olivier Roussel and Vasco Manquinho. “Pseudo-Boolean and cardinality constraints”. In: *Handbook of satisfiability*. IOS Press, 2021, pp. 1087–1129.

[9] Donald E Knuth. *The art of computer programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 2015.

[10] Gilles Audemard and Laurent Simon. “Refining restarts strategies for SAT and UNSAT”. In: *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*. Springer. 2012, pp. 118–126.

[11] Chu Min Li and Felip Manya. “MaxSAT, hard and soft constraints”. In: *Handbook of satisfiability*. IOS Press, 2021, pp. 903–927.

[12] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. “Maximum Satisfiabiliy”. In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 929–991.

[13] JP Marques Silva and Karem A Sakallah. “GRASP-a new search algorithm for satisfiability”. In: *Proceedings of International Conference on Computer Aided Design*. IEEE. 1996, pp. 220–227.

[14] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. “Preprocessing in SAT Solving.” In: *Handbook of satisfiability* 336 (2021), pp. 391–435.

[15] Niklas Eén and Niklas Sörensson. “Temporal induction by incremental SAT solving”. In: *Electronic Notes in Theoretical Computer Science* 89.4 (2003), pp. 543–560.

[16] Anton Belov and Joao Marques-Silva. “MUSer2: An efficient MUS extractor”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 8.3-4 (2012), pp. 123–128.

[17] Andreas Niskanen, Jeremias Berg, and Matti Järvisalo. “Incremental Maximum Satisfiability”. In: *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[18] Niklas Eén and Niklas Sörensson. “An extensible SAT-solver”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[19] Paul Shaw. “Using constraint programming and local search methods to solve vehicle routing problems”. In: *International conference on principles and practice of constraint programming*. Springer. 1998, pp. 417–431.

[20] David Pisinger and Stefan Ropke. “Large neighborhood search”. In: *Handbook of metaheuristics*. Springer, 2010, pp. 399–419.

[21] Randy Hickey and Fahiem Bacchus. “Large Neighbourhood Search for Anytime MaxSAT Solving”. In: ().

[22] Marek Piotrów. “Uwrmaxsat: Efficient solver for maxsat and pseudo-boolean problems”. In: *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2020, pp. 132–136.

[23] Milan Banković and David Šćepanović. “Trail Saving in SMT”. In: (2022).

[24] Alexander Nadel. “Introducing Intel (R) SAT Solver”. In: *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[25] Randy Hickey and Fahiem Bacchus. “Speeding up assumption-based SAT”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2019, pp. 164–182.

[26] Randy Hickey and Fahiem Bacchus. “Trail saving on backtrack”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2020, pp. 46–61.

[27] Stephen A Cook. “The complexity of theorem-proving procedures”. In: *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*. 2023, pp. 143–152.

[28] Joao Marques-Silva, Inês Lynce, and Sharad Malik. “Conflict-driven clause learning SAT solvers”. In: *Handbook of satisfiability*. ios Press, 2021, pp. 133–182.

[29] Matthew W Moskewicz et al. “Chaff: Engineering an efficient SAT solver”. In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 530–535.

[30] Hadi Katebi, Karem A Sakallah, and Joao P Marques-Silva. “Empirical study of the anatomy of modern SAT solvers”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2011, pp. 343–356.

[31] Armin Biere. “PicoSAT essentials”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 4.2-4 (2008), pp. 75–97.

[32] Gilles Audemard and Laurent Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers.” In: *IJCAI*. Vol. 9. 2009, pp. 399–404.

[33] Mikoláš Janota and Joao Marques-Silva. “On the query complexity of selecting minimal sets for monotone predicates”. In: *Artificial Intelligence* 233 (2016), pp. 73–83.

[34] Jessica Davies and Fahiem Bacchus. “Exploiting the power of MIP solvers in MAXSAT”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2013, pp. 166–181.

[35] Paul Saikko, Jeremias Berg, and Matti Järvisalo. “LMHS: a SAT-IP hybrid MaxSAT solver”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2016, pp. 539–546.

[36] Joao Marques-Silva and Jordi Planes. “On using unsatisfiability for solving maximum satisfiability”. In: *arXiv preprint arXiv:0712.1097* (2007).

[37] António Morgado, Carmine Dodaro, and Joao Marques-Silva. “Core-guided MaxSAT with soft cardinality constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2014, pp. 564–573.

[38] Fadi A Aloul et al. “Solving difficult SAT instances in the presence of symmetry”. In: *Proceedings of the 39th annual Design Automation Conference*. 2002, pp. 731–736.

[39] Joao Marques-Silva et al. “Boolean lexicographic optimization: algorithms & applications”. In: *Annals of Mathematics and Artificial Intelligence* 62 (2011), pp. 317–343.

[40] Alexander Nadel. “Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization”. In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2019, pp. 193–202. DOI: 10.23919/FMCAD.2019.8894273. URL: https://doi.org/10.23919/FMCAD.2019.8894273.

[41] Holger H Hoos and Thomas Stützle. “Local search algorithms for SAT: An empirical evaluation”. In: *Journal of Automated Reasoning* 24.4 (2000), pp. 421–481.

[42] Zhendong Lei et al. “SATLike-c: Solver description”. In: *MaxSAT Evaluation* 2021 (2021), p. 19.

[43] Gianpiero Cabodi et al. “Speeding-up heuristic allocation, scheduling and binding with SAT-based abstraction/refinement techniques”. In: *ACM Trans. Design Autom. Electr. Syst.* 15.2 (2010), 12:1–12:34. URL: https://doi.org/10.1145/1698759.1698762.

[44] Koen Claessen and Niklas Sörensson. “A liveness checking algorithm that counts”. In: *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*. 2012, pp. 52–59. URL: http://ieeexplore.ieee.org/document/6462555/.

[45] Niklas Eén, Alan Mishchenko, and Nina Amla. “A single-instance incremental SAT formulation of proof- and counterexample-based abstraction”. In: *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. 2010, pp. 181–188. URL: http://ieeexplore.ieee.org/document/5770948/.

[46] Niklas Eén and Niklas Sörensson. “Temporal induction by incremental SAT solving”. In: *Electr. Notes Theor. Comput. Sci.* 89.4 (2003), pp. 543–560. URL: https://doi.org/10.1016/S1571-0661(05)82542-3.

[47] Fahiem Bacchus and George Katsirelos. “Using Minimal Correction Sets to More Efficiently Compute Minimal Unsatisfiable Sets”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 2015, pp. 70–86. URL: https://doi.org/10.1007/978-3-319-21668-3_5.

[48] Anton Belov, Inês Lynce, and João Marques-Silva. “Towards efficient MUS extraction”. In: *AI Commun.* 25.2 (2012), pp. 97–116. URL: https://doi.org/10.3233/AIC-2012-0523.

[49] Mark H. Liffiton et al. “Fast, flexible MUS enumeration”. In: *Constraints* 21.2 (2016), pp. 223–250. URL: https://doi.org/10.1007/s10601-015-9183-0.

[50] Fahiem Bacchus et al. “Relaxation Search: A Simple Way of Managing Optional Clauses”. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.* 2014, pp. 835–841. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8618.

[51] Carlos Mencía, Alessandro Previti, and João Marques-Silva. “Literal-Based MCS Extraction”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. 2015, pp. 1973–1979. URL: http://ijcai.org/Abstract/15/280.

[52] Alessandro Previti et al. “Improving MCS Enumeration via Caching”. In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. 2017, pp. 184–194. URL: https://doi.org/10.1007/978-3-319-66263-3_12.

[53] Mario Alviano, Carmine Dodaro, and Francesco Ricca. “A MaxSAT Algorithm Using Cardinality Constraints of Bounded Size”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. 2015, pp. 2677–2683. URL: http://ijcai.org/Abstract/15/379.

[54] Carlos Ansótegui, Frédéric Didier, and Joel Gabàs. “Exploiting the Structure of Unsatisfiable Cores in MaxSAT”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. 2015, pp. 283–289. URL: http://ijcai.org/Abstract/15/046.

[55] Jessica Davies and Fahiem Bacchus. “Postponing Optimization to Speed Up MAXSAT Solving”. In: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*. 2013, pp. 247–262. URL: https://doi.org/10.1007/978-3-642-40627-0_21.

[56] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. “Open-WBO: A Modular MaxSAT Solver,” in: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. 2014, pp. 438–445. URL: https://doi.org/10.1007/978-3-319-09284-3_33.

[57] António Morgado, Carmine Dodaro, and João Marques-Silva. “Core-Guided MaxSAT with Soft Cardinality Constraints”. In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. 2014, pp. 564–573. URL: https://doi.org/10.1007/978-3-319-10428-7_41.

[58] Nina Narodytska and Fahiem Bacchus. “Maximum Satisfiability Using Core-Guided MaxSAT Resolution”. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.* 2014, pp. 2717–2723. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8513.

[59] Paul Saikko, Jeremias Berg, and Matti Järvisalo. “LMHS: A SAT-IP Hybrid MaxSAT Solver”. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 2016, pp. 539–546. URL: https://doi.org/10.1007/978-3-319-40970-2_34.

[60] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. 2003, pp. 502–518. URL: https://doi.org/10.1007/978-3-540-24605-3_37.

[61] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. “Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction”. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*. 2013, pp. 309–317. URL: https://doi.org/10.1007/978-3-642-39071-5_23.

[62] Armin Biere. “CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018”. In: *Proceedings of SAT COMPETITION 2018 Solver and Benchmark Descriptions*. Ed. by Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. University of Helsinki, 2018.

[63] Mate Soos. “The CryptoMiniSat 5.5 set of solvers at the SAT Competition 2018”. In: *Proceedings of SAT COMPETITION 2018 Solver and Benchmark Descriptions*. Ed. by Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. University of Helsinki, 2018.

[64] Jean-Marie Lagniez and Armin Biere. “Factoring Out Assumptions to Speed Up MUS Extraction”. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*. 2013, pp. 276–292. URL: https://doi.org/10.1007/978-3-642-39071-5_21.

[65] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. “Ultimately Incremental SAT”. In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. 2014, pp. 206–218. URL: https://doi.org/10.1007/978-3-319-09284-3_16.

[66] Alexander Nadel and Vadim Ryvchin. “Efficient SAT Solving under Assumptions”. In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. 2012, pp. 242–255. URL: https://doi.org/10.1007/978-3-642-31612-8_19.

[67] Jessica Davies and Fahiem Bacchus. “Solving MAXSAT by Solving a Sequence of Simpler SAT Instances”. In: *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*. 2011, pp. 225–239. URL: https://doi.org/10.1007/978-3-642-23786-7_19.

[68] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. “RC2: a Python-based MaxSAT Solver”. In: *MaxSAT Evaluation 2018 Solver and Benchmark Descriptions*. Ed. by Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. University of Helsinki, 2018.

[69] João P. Marques Silva, Inês Lynce, and Sharad Malik. “Conflict-Driven Clause Learning SAT Solvers”. In: *Handbook of Satisfiability*. IOS Press, 2009, pp. 131–153. URL: https://doi.org/10.3233/978-1-58603-929-5-131.

[70] Armin Biere et al. “Conflict-driven clause learning sat solvers”. In: *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* (2009), pp. 131–153.

[71] Ian P. Gent. “Optimal Implementation of Watched Literals and More General Techniques”. In: *J. Artif. Intell. Res.* 48 (2013), pp. 231–251. URL: https://doi.org/10.1613/jair.4016.

[72] Peter Van Der Tak, Antonio Ramos, and Marijn Heule. “Reusing the assignment trail in cdcl solvers”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2011), pp. 133–138.

[73] Jeremias Berg, Fahiem Bacchus, and Alex Poole. “Abstract cores in implicit hitting set MaxSat solving”. In: *Theory and Applications of Satisfiability Testing–SAT 2020: 23rd International Conference, Alghero, Italy, July 3–10, 2020, Proceedings 23*. Springer. 2020, pp. 277–294.

[74] Alexey Ignatiev. “MSE18 Benchmarks: DRMX-AtMostK”. In: *MaxSAT Evaluation 2018* (), p. 30.

[75] *MaxSat Evaluation Series: 2006-2016* http://www.maxsat.udl.cat/*, 2017-2018* https://maxsat-evaluations.github.io/.

[76] Aaron R Bradley and Zohar Manna. “Checking safety by inductive generalization of counterexamples to induction”. In: *Formal Methods in Computer Aided Design (FMCAD’07)*. IEEE. 2007, pp. 173–180.

[77] Daniel Le Berre and Anne Parrain. “The Sat4j library, release 2.2”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7.2-3 (2010), pp. 59–64.

[78] Johan de Kleer and Raymond Reiter. “Foundations for assumption-based truth maintenance systems: Preliminary report”. In: *Proc. American Assoc. for Artificial Intelligence Nat. Conf.* 1987, pp. 183–188.

[79] Niklas Sörensson and Armin Biere. “Minimizing learned clauses”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2009, pp. 237–243.

[80] Matthew W. Moskewicz et al. “Chaff: Engineering an Efficient SAT Solver”. In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001, pp. 530–535. URL: https://doi.org/10.1145/378239.379017.

[81] Chuan Jiang and Ting Zhang. “Partial Backtracking in CDCL Solvers”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*. Ed. by Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Springer, 2013, pp. 490–502. URL: https://doi.org/10.1007/978-3-642-45221-5_33.

[82] Alexander Nadel and Vadim Ryvchin. “Chronological Backtracking”. In: *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Vol. 10929. Lecture Notes in Computer Science. Springer, 2018, pp. 111–121. URL: https://doi.org/10.1007/978-3-319-94144-8_7.

[83] Sibylle Möhle and Armin Biere. “Backing Backtracking”. In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 250–266. URL: https://doi.org/10.1007/978-3-030-24258-9_18.

[84] Marijn J.H. Heule, Matti Järvisalo, and Martin Suda, eds. *Proc. of SAT Race 2019: Solver and Benchmark Descriptions*. University of Helsinki, 2019. URL: http://hdl.handle.net/10138/306988.

[85] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)”. In: *J. ACM* 53.6 (2006), pp. 937–977. URL: https://doi.org/10.1145/1217856.1217859.

[86] Alexander Nadel. “Understanding and Improving a Modern SAT Solver”. PhD thesis. Tel Aviv University, 2009.

[87] Maurice Bruynooghe. “Solving combinatorial search problems by intelligent backtracking”. In: *Information processing letters* 12.1 (1981), pp. 36–39.

[88] Christian Bliek. “Generalizing partial order and dynamic backtracking”. In: *AAAI/IAAI* 98 (1998), pp. 319–325.

[89] Matthew L Ginsberg. “Dynamic backtracking”. In: *Journal of artificial intelligence research* 1 (1993), pp. 25–46.

[90] David A McAllester. “Partial order backtracking”. In: *Journal of Artificial Intelligence Research* 1 (1993), pp. 17–24.

[91] Gilles Audemard and Laurent Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers”. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 399–404. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

[92] Matti Järvisalo, Marijn JH Heule, and Armin Biere. “Inprocessing rules”. In: *International Joint Conference on Automated Reasoning*. Springer. 2012, pp. 355–370.

[93] Clark Barrett et al. “Satisfiability modulo theories”. In: *Handbook of satisfiability*. IOS Press, 2021, pp. 1267–1329.

[94] Randy Hickey and Fahiem Bacchus. “Speeding Up Assumption-Based SAT”. In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 164–182. URL: https://doi.org/10.1007/978-3-030-24258-9_11.

[95] Marijn J. H. Heule, Matti Juhani Järvisalo, and Martin Suda, eds. *Proc. of SAT Competition 2018: Solver and Benchmark Descriptions*. University of Helsinki, 2018. URL: http://hdl.handle.net/10138/237063.

[96] Nils Froleyks et al. "SAT competition 2020". In: *Artificial Intelligence* 301 (2021), p. 103572.

[97] Mao Luo et al. "An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. Ed. by Carles Sierra. ijcai.org, 2017, pp. 703–711. URL: https://doi.org/10.24963/ijcai.2017/98.

[98] Fan Xiao et al. "Maplelrb lcm, maple lcm, maple lcm dist, maplelrb lcmoccrestart and glucose-3.0+ width in sat competition 2017". In: *Proc. of SAT Competition 2017: Solver and Benchmark Descrptions*. Ed. by Tomás Balyo, Marijn J. H. Heule, and Matti Juhani Järvisalo. University of Helsinki, 2017, pp. 22–23. URL: http://hdl.handle.net/10138/224324.

[99] Norbert Manthey. "The mergesat solver". In: *Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings 24*. Springer. 2021, pp. 387–398.

[100] Randy Hickey, Nick Feng, and Fahiem Bacchus. "Cadical-trail, Cadical-alluip, Cadical-alluip-trail, and Maple-LCM-Dist-alluip-trail at SAT Competition 2020". In: *SAT COMPETITION 2020* (2020), p. 10.

[101] Jussi Rintanen. "Planning and SAT". In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 765–789.

[102] Tomas Geffner and Hector Geffner. "Compact policies for fully observable non-deterministic planning as SAT". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 28. 1. 2018.

[103] Christian Muise, Sheila McIlraith, and Christopher Beck. "Improved non-deterministic planning by exploiting state relevance". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 22. 1. 2012.

[104] Daniel Bryce and Olivier Buffet. "6th international planning competition: Uncertainty part". In: *Proceedings of the 6th International Planning Competition (IPC'08)* (2008).

[105] Niklas Sorensson and Niklas Een. "Minisat v1. 13-a sat solver with conflict-clause minimization". In: *SAT* 2005.53 (2005), pp. 1–2.

[106] Armin Biere et al. *Handbook of satisfiability*. Vol. 336. IOS press, 2021.

[107] Armin Biere. *CaDiCaL simplified satisfiability solver*. 2020.

[108] Gilles Audemard and Laurent Simon. "On the glucose SAT solver". In: *International Journal on Artificial Intelligence Tools* 27.01 (2018), p. 1840001.

[109] V Ganesh and JH Liang. *MapleSAT*. 2017.

[110] Jussi Rintanen. "Planning as satisfiability: Heuristics". In: *Artificial intelligence* 193 (2012), pp. 45–86.

[111] SEPARATE DECISION QUEUE. "CADICAL at the SAT Race 2019". In: *SAT RACE 2019* (), p. 8.

[112] Armin Biere. "Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018". In: *Proceedings of SAT Competition* (2017), pp. 14–15.

[113] Chu Min Li and Felip Manyà. “MaxSAT, Hard and Soft Constraints”. In: *Handbook of Satisfiability*. IOS Press, 2009, pp. 613–631. DOI: 10.3233/978-1-58603-929-5-613. URL: http://dx.doi.org/10.3233/978-1-58603-929-5-613.

[114] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. “Maximum Satisfiability”. In: *Handbook of Satisfiability*. IOS Press, 2020.

[115] Emir Demirović and Nysret Musliu. “MaxSAT-based large neighborhood search for high school timetabling”. In: *Computers & Operations Research* 78 (2017), pp. 172–180.

[116] Jeremias Berg, Emir Demirović, and Peter J Stuckey. “Core-boosted linear search for incomplete MaxSAT”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2019, pp. 39–56.

[117] Shaowei Cai and Zhendong Lei. “Old techniques in new ways: Clause weighting, unit propagation and hybridization for maximum satisfiability”. In: *Artificial Intelligence* 287 (2020), p. 103354.

[118] Shaowei Cai et al. “New local search methods for partial MaxSAT”. In: *Artificial Intelligence* 240 (2016), pp. 1–18.

[119] Mireille Palpant, Christian Artigues, and Philippe Michelon. “LSSPER: Solving the resource-constrained project scheduling problem with large neighbourhood search”. In: *Annals of Operations Research* 131 (2004), pp. 237–257.

[120] Jari Kytöjoki et al. “An efficient variable neighborhood search heuristic for very large scale vehicle routing problems”. In: *Computers & operations research* 34.9 (2007), pp. 2743–2757.

[121] Burton H Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[122] Fahiem Bacchus. “MaxHS in the 2020 MaxSat Evaluation”. In: *MaxSAT Evaluation 2020* (2020), p. 19.

[123] Fahiem Bacchus et al. *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*. University of Helsinki, Department of Computer Science, 2020.

[124] Michele Lombardi and Pierre Schaus. “Cost impact guided LNS”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2014, pp. 293–300.

[125] Sophie N Parragh and Verena Schmid. “Hybrid column generation and large neighborhood search for the dial-a-ride problem”. In: *Computers & Operations Research* 40.1 (2013), pp. 490–497.

[126] Pavlos S Efraimidis and Paul G Spirakis. “Weighted random sampling with a reservoir”. In: *Information Processing Letters* 97.5 (2006), pp. 181–185.

[127] Laurent Perron, Paul Shaw, and Vincent Furnon. “Propagation guided large neighborhood search”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2004, pp. 468–481.

[128] Pierre Hansen, Nenad Mladenović, and Jose A Moreno Perez. “Variable neighbourhood search: methods and applications”. In: *Annals of Operations Research* 175 (2010), pp. 367–407.

[129] Andreas Niskanen, Jeremias Berg, and Matti Järvisalo. “Enabling Incrementality in the Implicit Hitting Set Approach to MaxSAT under Changing Weights”. In: *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2021.

[130] Chu Min Li et al. “Combining clause learning and branch and bound for MaxSAT”. In: (2021).

[131] Alexander Nadel. “TT-Open-WBO-Inc-21: an Anytime MaxSAT Solver Entering MSE’21”. In: *MaxSAT Evaluation 2021* (2021), pp. 21–22.

[132] Jeremias Berg, Emir Demirović, and Peter J Stuckey. “Loandra in the 2020 MaxSAT Evaluation”. In: *MaxSAT Evaluation 2021* (2021), pp. 24–25.

[133] Zhendong Lei et al. “SATLike-c: Solver Description”. In: *MaxSAT Evaluation 2021* (2021), p. 19.

[134] Tuukka Korhonen et al. “MaxPre: an extended MaxSAT preprocessor”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2017, pp. 449–456.

[135] Aviad Cohen, Alexander Nadel, and Vadim Ryvchin. “Local Search with a SAT Oracle for Combinatorial Optimization”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2021, pp. 87–104.

[136] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. “MaxSAT-Based Postprocessing for Treedepth”. In: *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*. Ed. by Helmut Simonis. Vol. 12333. Lecture Notes in Computer Science. Springer, 2020, pp. 478–495. DOI: 10.1007/978-3-030-58475-7_28. URL: https://doi.org/10.1007/978-3-030-58475-7_28.

[137] Gustav Björdal et al. “Solving satisfaction problems using large-neighbourhood search”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2020, pp. 55–71.

[138] Stefan Ropke and David Pisinger. “An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows”. In: *Transportation science* 40.4 (2006), pp. 455–472.

[139] Ruben Martins, Matti Jarvisalo, and Fahiem Bacchus. “MaxSAT Evaluation 2018”. In: *SAT* 2018 (2018).

[140] Marijn JH Heule and Armin Biere. “Proofs for satisfiability problems”. In: *All about Proofs, Proofs for all* 55.1 (2015), pp. 1–22.