amU
Aix Marseille Université

NNT : 2024AIXM0001

THÈSE DE DOCTORAT

Soutenue à AMU — Aix-Marseille Université
le 16 décembre 2024 par

Shuolin Li

Exact solving of SAT and MaxSAT

Discipline
Informatique

École doctorale
ED 184 Mathèmatiques et Informatique

Laboratoire/Partenaires de recherche
Laboratoire d'Informatique et Systèmes

LABORATOIRE
D'INFORMATIQUE
& SYSTÈMES
UMR 7020

Composition du jury

Carlos Ansótegui Université de Lleida, Espagne	Rapporteur
Gilles Audemard Université d'Artois, France	Rapporteur
Christine Solnon INSA de Lyon, France	Présidente
Chu-Min Li Université de Picardie Jules Verne, France	Directeur de thèse
Djamal Habet Université d'Aix-Marseille, France	Co-directeur de thèse

Affidavit

I, undersigned, Shuolin Li, hereby declare that the work presented in this manuscript is my own work, carried out under the scientific supervision of Chu-Min Li and Djamal Habet, in accordance with the principles of honesty, integrity and responsibility inherent to the research mission. The research work and the writing of this manuscript have been carried out in compliance with both the french national charter for Research Integrity and AMU charter on the fight against plagiarism.

This work has not been submitted previously either in this country or in another country in the same or in a similar version to any other examination body.

Marseille, 2024 October 10



This work is licensed under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International Public License

List of Publications

International Journals

1. Mao Luo, Chumin Li, Xinyun Wu, Shuolin Li, Zhipeng Lv. Branching Strategy Selection Approach Based on Vivification Ratio. International Journal of Software and Informatics, 2022,12(1):131 151

International Conferences

1. Shuolin Li, Chu-Min Li, Mao Luo, Jordi Coll, Djamal Habet, and Felip Manyà. A New Variable Ordering for In-processing Bounded Variable Elimination in Sat Solvers. In the Proceedings of the 32th International Joint Conferences on Artificial Intelligence (IJCAI 2023). Macao, China, August 19-25, 2023.

2. Shuolin Li, Jordi Coll, Djamal Habet, Chu-Min Li, and Felip Manyà. A Tableau Calculus for Signed Maximum Satisfiability. In the Proceedings of the 53rd International Symposium on Multiple-Valued Logic (ISMVL 2023). Matsue, Japan, May 22-24, 2023.

3. Shuolin Li, Jordi Coll, Djamal Habet, Chu-Min Li, and Felip Manyà. A Tableau Calculus for Maxsat Based on Resolution. In Artificial Intelligence Research and Development, IOS Press (CCIA 2022). Sitges, Spain, October 19-21, 2022.

International Workshops

1. Shuolin Li, Chu-Min Li, Jordi Coll, Djamal Habet, and Felip Manyà. Get the Better Cores through Unlocking Mechanism in Unit Propagation. In Doctoral Program of the International Conference on Principles and Practice of Constraint Programming (CP 2024). Girona, Spain, September 02-06, 2024.

2. Shuolin Li. Guiding Problem Simplification with Variable Activities in SAT Solvers. In Doctoral Consortium at ECAI-2024. Santiago de Compostela, Spain, October 19-24, 2024.

Résumé

Au cœur de l'informatique et de l'intelligence artificielle, la logique sert de cadre fondamental pour la modélisation et la résolution d'un large éventail de problèmes complexes, tant en théorie que dans les applications pratiques. Un formalisme important dans ce domaine est le problème de satisfaction (SAT), qui demande si une formule propositionnelle donnée, généralement représentée comme un ensemble de contraintes appelées clauses, peut être satisfaite. Une extension d'optimisation naturelle du SAT est le problème de satisfaction maximale (MaxSAT), où l'objectif est de trouver une affectation qui satisfait le nombre maximal possible de clauses dans la formule.

SAT et MaxSAT ont tous deux des applications étendues dans des domaines tels que la vérification, la planification, la conception de circuits, la bioinformatique et l'intelligence artificielle. Ces problèmes sont fondamentaux dans divers domaines et bénéficient souvent des améliorations des techniques algorithmiques employées par les solveurs SAT et MaxSAT.

Dans notre travail, nous nous concentrons sur l'étude des limites des solveurs SAT et MaxSAT existants et sur l'amélioration de leur efficacité. Notre première contribution consiste à guider les simplifications de formules par des scores heuristiques variables. Cette approche intègre la recherche de solutions et les simplifications de formules dans les solveurs SAT, leur permettant de se renforcer mutuellement et d'améliorer l'efficacité globale. De plus, nous améliorons la méthode d'estimation de la borne inférieure dans les solveurs Branch and Bound MaxSAT grâce au mécanisme de déverrouillage, qui accélère le processus d'élagage et conduit à une convergence plus rapide vers des solutions optimales.

En plus de ces contributions pratiques, nous étendons également nos recherches au système de preuve MaxSAT. Nous améliorons le calcul MaxSAT Tableau existant en introduisant de nouvelles règles d'inférence et en étendant le calcul pour prendre en charge MaxSAT signé, enrichissant ainsi le cadre théorique et élargissant son applicabilité.

Mots clés : Intelligence Artificielle, SAT, Élimination de Variables Bornées, MaxSAT, Branche et Lié, Calcul des Tableaux

Abstract

At the core of computer science and artificial intelligence, logic serves as a foundational framework for modeling and solving a wide range of complex problems, both in theory and in practical applications. A prominent formalism in this domain is the Satisfiability (SAT) problem, which asks whether a given propositional formula, typically represented as a set of constraints known as clauses, can be satisfied. A natural optimization extension of SAT is the Maximum Satisfiability (MaxSAT) problem, where the objective is to find an assignment that satisfies the maximum possible number of clauses in the formula.

Both SAT and MaxSAT have borad applications in areas such as verification, planning, circuit design, bioinformatics and artificial intelligence. These problems are fundamental in various fields and are often benifited from the improvements of algorithmic techniques employed by SAT and MaxSAT solvers.

In our work, we focus on investigating the limitations of existing SAT and MaxSAT solvers and enhancing their effectiveness. Our first contribution lies in guiding the formula simplifications by variable heuristic scores. This approach integrates solution search and formula simplifications in SAT solvers, allowing them to reinforce each other and improve overall efficiency. Moreover, we enhance the lower bound estimation method in Branch and Bound MaxSAT solvers through the unlocking mechanism, which accelerates the pruning process and leads to faster convergence on optimal solutions.

In addition to these practical contributions, we also extend our research to the MaxSAT proof system. We improve the existing MaxSAT Tableau calculus by introducing new inference rules and extending the calculus to accommodate signed MaxSAT, thus enriching the theoretical framework and broadening its applicability.

Keywords: Artificial Intelligence, SAT, Bounded Variable Elimination, MaxSAT, Branch and Bound, Tableau Calculus

Acknowledgement

Contents

Affidavit 2

List of Publications 3

Abstract 5

Acknowledgement 6

Contents 7

List of Figures 10

List of Tables 12

Introduction 14

1 Preliminaries 17
- 1.1 Complexity Theory . . . 17
- 1.2 Deductive Logic and Propositional Logic . . . 20
  - 1.2.1 Deductive Logic . . . 20
  - 1.2.2 Propositional Logic . . . 21
- 1.3 Conclusion . . . 22

2 Satisfiability 24
- 2.1 Definition and Variants . . . 24
- 2.2 Complete Algorithms . . . 26
  - 2.2.1 The DP Algorithm . . . 26
  - 2.2.2 The DPLL Algorithm . . . 29
  - 2.2.3 The CDCL Algorithm . . . 35
- 2.3 Incomplete Algorithm . . . 42
- 2.4 Simplification Algorithms . . . 46
  - 2.4.1 Bounded Variable Elimination . . . 46
  - 2.4.2 Failed, Implied, Equivalent Literals Detection . . . 50
  - 2.4.3 Clause Garbage Collection . . . 50
- 2.5 Conclusion . . . 51

3 Maximum Satisfiability 52
- 3.1 Definition and Variants . . . 52

3.2 Branch-and-Bound Algorithm . . . 53
3.2.1 Branch Heuristic . . . 55
3.2.2 Inference rule . . . 56
3.2.3 Lower Bound Estimation . . . 59
3.3 SAT-based Algorithms . . . 62
3.3.1 Model-improving . . . 62
3.3.2 Core-guided . . . 62
3.3.3 Implicit Hitting Set . . . 62
3.4 Incomplete Algorithm . . . 63
3.5 Conclusion . . . 67

4 A New Variable Ordering for In-processing Bounded Variable Elimination in SAT Solvers 69
4.1 Introduction & Motivation . . . 69
4.2 ESA: Variable Elimination Scheduled by Activity . . . 71
4.2.1 The Principle of ESA . . . 71
4.2.2 Implementation of ESA in Cadical and Kissat Based Solvers . . . 73
4.3 Experimental Results . . . 74
4.3.1 Effectiveness of the ESA Procedure . . . 74
4.3.2 Analyzing the Behaviour of ESA . . . 77
4.4 Other Implementations . . . 79
4.4.1 Keep Binary Learnt Clauses after Variable elimination . . . 79
4.4.2 Variable Heuristic in Other Simplification Functions . . . 79
4.5 Solvers at SAT Competitions . . . 81
4.6 Conclusion . . . 82

5 Improving the Lower Bound in Branch-and-Bound Algorithms for MaxSAT 84
5.1 Motivation . . . 84
5.2 Lookahead with Unlocking Mechanism for Unweighted MaxSAT . . . 85
5.3 Lookahead with Unlocking Mechanism for Weighted MaxSAT . . . 86
5.4 Experimental Evaluation . . . 90
5.4.1 Experimental settings . . . 90
5.4.2 Impact of the unlocking mechanism . . . 91
5.4.3 Empirical analysis . . . 91
5.5 Conclusions and Future Work . . . 93

6 Tableau Calculus for MaxSAT 97
6.1 Preliminaries . . . 98
6.1.1 Boolean logic . . . 98
6.1.2 Signed logic . . . 99
6.1.3 Tableau calculi for SAT . . . 100
6.2 Related Work . . . 101
6.2.1 Tableau calculi for clausal MaxSAT . . . 101

6.2.2 Tableau calculi for non-clausal MaxSAT 103
6.3 A Tableau Calculus for MaxSAT based on Resolution 104
6.3.1 Soundness and Completeness 107
6.3.2 Extending to the Weighted Partial MaxSAT 109
6.4 A Tableau Calculus for Signed MaxSAT 111
6.4.1 Soundness and Completeness 112
6.4.2 Extending to the Weighted Partial Signed MaxSAT 114
6.5 Conclusion 114

Conclusion **116**

Bibliography **117**

List of Figures

1.1 Euler diagram on the implications of P≠NP (left) and P=NP (right) . . . 20

2.1 The solving process of DP algorithm . . . 29
2.2 The process of backtracking . . . 32
2.3 Execution of the DPLL algorithm . . . 34
2.4 The implication graph . . . 36
2.5 The implication graph with First-UIP cut . . . 37
2.6 The implication graph with Last-UIP cut . . . 37

3.1 Execution of a BnB algorithm for Max-SAT . . . 56
3.2 An example of implication graph to identify a core. The formula ϕ contains 3 soft clauses: (s_1), (s_2), (s_3), and 4 hard clauses: $(\neg s_1, x_1), (\neg x_1, s_2), (\neg s_2, \neg x_2)$ and $(\neg s_3, \neg x_1, x_2)$. Each node represents a satisfied literal l. With the incoming edges, it represents also the hard clause allowing to propagate l. For example, the node x_2 represents the clause $(\neg s_3, \neg x_1, x_2)$. Propagating (s_1) satisfies x_1, s_2 and $\neg x_2$, but propagating (s_3) satisfies x_2, making $(\neg s_2, \neg x_2)$ or $(\neg s_3, \neg x_1, x_2)$ empty. By retracing the implication graph from the empty clause to the propagated soft clauses, we know that (s_1) and (s_3) cannot be satisfied at the same time. So, the core $\{(s_1), (s_3)\}$ is identified. Note that s_2 is propagated because of the hard clause $(\neg x_1, s_2)$ and does not belong to the identified core. . . . 60

4.1 The resolution tree of regular resolution. From the leaf nodes to the root node, the algorithm performs regular resolution on the variables $x_1, \ldots, x_6, x$ in order. The pivot variable at each resolution are marked red. As can be seen, no variables are reused, and it takes 11 steps for the algorithm to prove the unsatisfiability. . . . 71
4.2 The resolution tree of unrestricted resolution. The variable x is used twice on the path to the root node, it is marked red. Only 10 steps are needed for the algorithm to prove the unsatisfiability. . . . 71
4.3 Cactus plots of different solvers. Each point (N, T) in a curve indicates the number N of instances solved within T secs by the corresponding solver. Recall that the frequence of BVE and decision heuristics are different in Cadical and KM. . . . 76
4.4 Scatter plot of the number of eliminated variables (left), and scatter plot of the number of learnt clauses lost per variable (right) by the 1st BVE of Kissat_MAB_ESA and Kissat_MAB. . . . 77

4.5 Scatter plot of the average number of eliminated variables per BVE call (above), and scatter plot of the average number of learnt clauses removed per variable among the first k BVE calls (below). 78
4.6 Anniversary main track results of the SAT Competition 2022 82

5.1 The process of normal core detection, soft literals s_1, s_2, s_3 are propagated 85
5.2 The process of normal core detection, soft literals s_1, s_4, s_5 are propagated 85
5.3 Number of instances solved with x seconds 92

6.1 Completed clausal SAT tableau (left) and completed clausal MaxSAT tableau (right) for the multiset of clauses $\phi = \{\neg x_1, \neg x_2, \neg x_3, x_1 \vee x_2, \neg x_1 \vee x_3\}$. 102
6.2 A non-clausal MaxSAT tableau for $\{x_1, x_2, \neg x_1 \wedge \neg x_2\}$. 104
6.3 A tableaux for a MaxSAT instance $\{(x_1), (x_2), (\neg x_1 \vee x_3), (\neg x_1 \vee \neg x_2 \vee \neg x_3)\}$.105
6.4 Tableau expansion rules for non-clausal MaxSAT 106
6.5 A tableaux for the non-clausal MaxSAT instance $\{x_1, x_2, \neg x_1 \vee x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3\}$. 107
6.6 Tableau expansion rules for Weighted Partial MaxSAT 110
6.7 A tableau for the WPMaxSAT instance $\{(\neg x_1 \vee \neg x_2, \top), (\neg x_2 \vee \neg x_3, 2), (x_1, 2), (x_2, 3), (x_3, 2)\}$ 111

List of Tables

1.1 Truth tables of propositional connectives 21

4.1 Results of Solvers with and without ESA 75
4.2 Results of Variant Solvers 77

5.1 The number of instances solved by (W)MaxCDCL with and without the unlocking mechanism. 96
5.2 Results for benchmarks of the unweighted track. 96
5.3 Results for benchmarks of the weighted track. 96
5.4 Percentage range of soft conflicts (soft confl), number of instances for each range (#inst), percentage of instances for each range (#inst/sum), success rate for stage 2 lookahead (stage2 succ), additional instances solved thanks to unlocking mechanism for each range (#unlock solved+). 96

6.1 α-formulas and β-formulas. 100
6.2 Tableau expansion rules for SAT 101
6.3 n-ary β-rule 101
6.4 Tableau expansion rules for non-clausal MaxSAT 103
6.5 Tableau expansion rules for Signed MaxSAT 112
6.6 Tableau expansion rules for Weighted Partial Signed MaxSAT 115

List of Algorithms

1 DP(ϕ), the SAT algorithm based on regular resolution 28
2 $DPLL(\phi, \alpha)$, a search algorithm 33
3 Analyse($learnt$, d), conflict analysis using First-UIP schema 38
4 CDCL(ϕ), the search algorithm in modern SAT solvers 43
5 SLS(ϕ), the general stochastic local search algorithm 44
6 BVE($\phi, L, O, growth, length$), a generic BVE procedure for CDCL solvers 49
7 LiteralsDetection($V_{unassigned}$), a generic procedure for failed, implied, equivalent literals detection 51
8 BnB(ϕ), the Branch and Bound algorithm in MaxSAT 55
9 LocalSearch(ϕ), the local search algorithm in MaxSAT 64
10 Kissat(ϕ), the simplified solving process of the kissat solver 70
11 Lookahead(H, S, α, UB), the algorithm to compute LB 94
12 WLookahead(H, (S, w), α, UB), the algorithm to compute LB for weighted MaxSAT 95

Introduction

The Boolean Satisfiability Problem (SAT) [19] is a cornerstone of theoretical computer science and mathematical logic, originally formalized as the problem of determining whether a Boolean formula can be satisfied, i.e., whether there exists an assignment of truth values to variables that makes the entire formula true. Since being established as the first NP-complete problem in 1971 [58], SAT has become a fundamental problem in complexity theory, with wide-ranging applications in areas such as artificial intelligence, formal verification, and cryptography. Building upon SAT, the Maximum Satisfiability Problem (MaxSAT) [10, 101] generalizes the challenge by seeking an assignment that satisfies the maximum number of clauses in a given formula, which is especially useful in many real world applications where complete satisfiability is impossible. MaxSAT has become essential in optimization problems across various domains, including planning, scheduling, and bioinformatics, and has led to significant research in algorithm design, particularly in the development of efficient solvers. Together, SAT and MaxSAT play a critical role in both theoretical research and practical applications, making them central to the study of computational complexity and combinatorial optimization.

The exact algorithms used to solve SAT and MaxSAT, which guarantee the determination of satisfiability or optimal solutions, have evolved significantly, leading to highly efficient techniques. For SAT, the most influential algorithm nowadays is the Conflict-Driven Clause Learning (CDCL) algorithm [140, 165], which is an extension of the classical Davis-Putnam-Logemann-Loveland (DPLL) algorithm [61]. CDCL combines non-chronological search which allows the solver to backtrack to a conflict-relevant decision level and enables the pruning of large portions of the search space, with learning from conflicts, which prevents the solver from being trapped into the same conflict in the future, making it highly effective for solving practical problems. For MaxSAT, it can be broadly categorized into branch-and-bound (BnB) [101] and SAT-based approaches [10]. The former systematically explore the solution space by branching on variable assignments and using bounds to prune subspaces that cannot find a better solution, ensuring an optimal solution is found while minimizing search time. The latter leverage efficient SAT solvers by transforming the MaxSAT problem into a series of SAT instances and iteratively solving them.

Our contributions in this manuscript revolve around the two problems, focusing on both practical solver performance and theoretical advancements.

Our first contribution falls within the scope of improving the performance of modern SAT solvers. We propose a method that uses variable heuristic scores to guide the simplification of formulas in SAT solvers. This approach integrates the search for solutions with simplifications, making the solver more efficient.

Our second contribution focuses on understanding the limitations of the core detection process in current Branch and Bound MaxSAT solvers. We introduce an unlocking mechanism in such solvers to improve the estimation of the lower bound. This accelerates the pruning process and leads to faster convergence toward optimal solutions.

At last, we contribute to the theoretical advancement of MaxSAT by improving the MaxSAT Tableau calculus. We extend the existing proof system to accommodate signed MaxSAT and introduce new inference rules, broadening the framework's applicability and enriching its theoretical foundations.

Our work has been published in international journal (IJSI) [135], international conferences (IJCAI, ISMVL, CCIA) [123, 118, 116], and international workshop (Doctoral Program in CP, Doctoral Consortium in ECAI) [120, 117]. The other publications include our solver descriptions submitted to the SAT competitions [119, 124, 125] and MaxSAT evaluations [112, 57, 113, 121]. For SAT, our solver Kissat_MAB_ESA won the gold and silver medals respectively in the Anniversary track and Anniversary UNSAT track of the SAT competitions 2022. For MaxSAT, our solvers MaxCDCL2023 and WMaxCDCL2023 won the silver and gold medals respectively in the exact unweighted track and exact weighted track of the MaxSAT evaluation 2023, and solvers MaxCDCL2024 and WMaxCDCL2024 won the silver and gold medals respectively in the exact unweighted track of the MaxSAT evaluation 2024.

The manuscript is organized into two parts. The first part, consisting of the first three chapters, provides the essential background on the problems researched in this work. Chapter 1 reviews the fundamental definitions and concepts associated with complexity theory and propositional logic. Chapters 2 and 3 offer comprehensive overviews of SAT and MaxSAT, respectively. We introduce the formal definitions, the variants and the applications of both problems, and the main algorithms used to solve them, which include both exact and local search approaches. The second part presents our contributions and is also organized into three chapters. Chapter 4 introduces our work using variable heuristic scores to guide different formula simplification methods. In Chapter 5 we study the core unlocking mechanism employed in BnB solvers, to improve the quality of lower bound and accelerate pruning. And in Chapter 6, we improve the current MaxSAT tableau calculus by introducing new reference rules, and extend it to signed MaxSAT.

Background

1 Preliminaries

Sommaire

1.1 Complexity Theory 17
1.2 Deductive Logic and Propositional Logic 20
1.2.1 Deductive Logic 20
1.2.2 Propositional Logic 21
1.3 Conclusion 22

In this Chapter, we present preliminary notions necessary for reading this manuscript. In Section 1.1 we recall major notions in complexity theory, including time complexity, decision and optimization problem, P and NP, et al. And in Section 1.2, we review the concepts in propositional logic and the components of the Conjunctive Normal Form (CNF) formula which will be used in the following chapters.

1.1 Complexity Theory

Complexity theory is a branch of theoretical computer science that studies the resources required to solve computational problems. These resources typically include time (how many steps it takes to solve a problem) and space (how much memory is needed). The goal of complexity theory is to classify problems based on the difficulty of solving them, and to understand the limits of efficient computation.

There are some key concepts in complexity theory:

Definition 1.1 (Computational Problem) *A computational problem is a question or task that a computer must solve, given an input. For example, determining whether a number is prime is a computational problem.*

Definition 1.2 (Big-O) *Big-O is a mathematical way to describe the upper bound of the time complexity or space complexity of an algorithm. It provides an asymptotic analysis of how the performance (in terms of time or memory usage) of an algorithm grows relative to the input size n, as n becomes large. Essentially, it helps to quantify how the runtime or resource consumption of an algorithm scales with larger inputs.*

For example:

$O(1)$ means the algorithm's running time is constant and does not depend on the input size.

$O(n)$ means the algorithm's running time grows linearly with the input size n.

$O(n^2)$ means the algorithm's running time grows quadratically, proportional to the square of the input size. $O(n^c)$ corresponds to cubic time. For $\forall c > 1, O(n^c)$ is polynomial time.

$O(\log(n))$ means the algorithm's running time grows logarithmically with the input size.

$O(2^n)$ means the algorithm's running time doubles with each additional input element, or it grows exponentially with the input size.

Definition 1.3 (Time Complexity) *The amount of time (number of steps) it takes for an algorithm to solve a problem, as a function of the input size n. This is usually measured using Big-O notation in definition 1.2.*

Definition 1.4 (Space Complexity) *The amount of memory space required by an algorithm as a function of the input size n.*

Definition 1.5 (Decision Problem) *A special type of computational problem where the answer is either "yes" or "no."*

Definition 1.6 (Optimization Problem) *An optimization problem is a problem where the goal is to find the best solution from a set of possible solutions, based on some optimization criteria (e.g., minimum, maximum, shortest path, lowest cost).*

Both decision problems and optimization problems are central concepts in complexity theory. Complexity theory classifies these types of problems based on the time required to solve them, and it explores their relationships with various complexity classes such as P (Polynomial Time), NP (Nondeterministic Polynomial Time), and EXP (Exponential Time).

Definition 1.7 (P) *The class of problems that can be solved by a deterministic Turing machine in polynomial time. That is, there exists an algorithm that solves the problem in time bounded by a polynomial function of the input size n (e.g., $O(n^2)$,$O(n^3)$).*

Definition 1.8 (NP) *The class of problems for which a solution, once given, can be verified in polynomial time by a deterministic machine. Or the class of problems that can be solved by a nondeterministic Turing machine in polynomial time. The two definitions are equivalent.*

Definition 1.9 (EXP) *The class of decision problems that can be solved by a deterministic Turing machine in exponential time, i.e., time bounded by $2^{p(n)}$, where $p(n)$ is a polynomial function of the input size n.*

The concept of polynomial time algorithms was introduced in [64]. The author argued that an algorithm should be considered "efficient" or "tractable" if it runs in polynomial time, leading to the formalization of the complexity class P [54]. Meanwhile, there were some research on EXP and its role in complexity hierarchies. Then, NP was formalized in [58]. Proposition 1.1 shows the relationship between them.

Proposition 1.1 $P \subseteq NP \subseteq EXP$.

Polynomial-time reduction is a fundamental concept in computational complexity theory. It is a method for relating the complexity of two problems by showing how one problem can be transformed into another in polynomial time. The idea is used to classify problems based on their difficulty. For example, NP-Hard problems and NP-Complete problems, as defined below.

Definition 1.10 (Polynomial-Time Reduction) *A problem A is polynomial-time reducible to another problem B, written as $A \leq_P B$, if there exists a polynomial-time algorithm (a deterministic Turing machine) that transforms any instance of problem A into an instance of problem B, such that the solution to the instance of B can be used to solve the corresponding instance of A. This process is called polynomial-time reduction, and we say A is reduced to B.*

Definition 1.11 (NP-Hard) *A decision problem B is NP-Hard if $\forall A \in NP, A \leq_P B$.*

Definition 1.12 (NP-Complete) *A decision problem B is NP-complete if $B \in NP$ and is NP-hard.*

NP-Hard problems may or may not be in NP themselves. But a NP-Complete problem is definetely a NP problem, and it is the hardest problem in NP, every other NP problem can be reduced to it in polynomial time. This means that if you could solve an NP-Complete problem in polynomial time, you could solve all NP problems in polynomial time (P=NP). The P=NP problem, first proposed in [58], which consists in proving or disproving P=NP, is still an open problem nowadays [1]. Figure 1.1 presents the relationships between P, NP, NP-Complete and NP-Hard.

[1] It's one of the Millennium Prize problems selected by the Clay Mathematics Institute, for more details please check https://www.claymath.org/millennium-problems

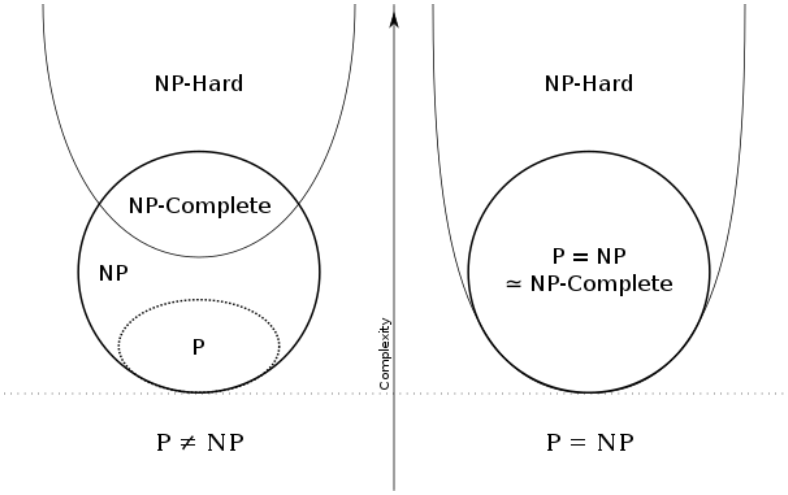


Figure 1.1: Euler diagram on the implications of P≠ NP (left) and P=NP (right) [2]

1.2 Deductive Logic and Propositional Logic

1.2.1 Deductive Logic

Deductive logic is a branch of logic that involves reasoning from general premises to reach a specific, logically certain conclusion. It is often contrasted with inductive logic, where reasoning moves from specific observations to general conclusions. The deductive reasoning composed by one or more general premises and a conclusion. It is usually written as follows, with the premises are upper the horizontal line and the conclusion below.

$$\frac{\begin{array}{c}\text{All humans are mortal.}\\ \text{Socrates is a human.}\end{array}}{\text{Socrates is mortal.}}$$

There are some concepts in deductive logic [18]:

Definition 1.13 (Argument) *An argument is a set of two or more sentences, one of which is the conclusion and the others are the premises.*

Definition 1.14 (Logical Validity) *An argument is logically valid if and only if it is not possible for the premises to be true and the conclusion false. An argument is logically invalid if and only if it is not logically valid.*

Definition 1.15 (Logical Soundness) *An argument is logically sound if and only if it is logically valid and all its premises are true. An argument is logically unsound if and only if it is not logically sound.*

[2] Figure made by Behnam Esfahbod, available under the Creative Commons Licence Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0).

Definition 1.16 (Logical Truth) *A sentence is logically true if and only if it is not possible for the sentence to be false.*

Definition 1.17 (Logical Falsity) *A sentence is logically false if and only if it is not possible for the sentence to be true.*

1.2.2 Propositional Logic

Propositional Logic (Sentential Logic) is a subset of deductive logic that deals with propositions (statements that can be true or false). It is defined on the following alphabet:

- **Propositional (Boolean) Variables**: variables can only take the Boolean values True or False, which are presented by English characters such as x, y.
- **Propositional Connectives**: include $\neg$ (negation), $\vee$ (conjunction), $\wedge$ (disjunction), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence).
- **Immediate Components**: the one or two components near the connectives.
- **Priority Symbols**: '(' and ')', the content in parentheses will be taken into account first.

We usually use letters ϕ to represent the propositional formula, its recursive definition is as follows:

Definition 1.18 (Propositional formula) *Let V be a set of propositional variables.*

- *Any variable $x \in V$ is a propositional formula.*
- *If x and y are propositional formulas, then (x), $\neg x$, $x \vee y$, $x \wedge y$, $x \Rightarrow y$ and $x \Leftrightarrow y$ are also propositional formulas.*

Example 1.1 *Given the set of variables $V = \{x_1, x_2, x_3, x_4\}$, $\phi = (\neg x_1 \vee x_2) \Leftrightarrow (x_3 \Rightarrow x_4)$ is a propositional formula.*

Just as propositional variables, the propositional formula also takes value true or false, and its value is decided by the value(s) of its immediate component(s), as shown in Table 1.1. We use F to represent False and T to represent True.

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \Rightarrow y$	$x \Leftrightarrow y$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

Table 1.1: Truth tables of propositional connectives

Next, we introduce some definitions needed in the discussion in future chapters. Given a set of variables V, we have:

Definition 1.19 (Literal) *A literal l is either a variable x or its negation $\neg x$.*

Definition 1.20 (Clause) *A clause cl is the disjunction of literals, $cl = (l_1 \vee ... \vee l_n)$, and we can also use set to represent it, i.e., $cl = \{l_1, ...l_n\}$.*

Definition 1.21 (CNF Formula) *A Conjunctive Normal Form (CNF) formula ϕ is the conjunction of clauses, $\phi = cl_1 \wedge ... \wedge cl_m$, or $\phi = \{cl_1, ..., cl_m\}$.*

Definition 1.22 (Assignment) *Let V be a set of propositional variables. An assignment of variables in V is a function α that maps $x \in V$ to value True or False. In practice, an assignment can be represented by a set of literals assigned true. For example, for an assignment $\alpha = \{l_1, \neg l_2\}$, the assignment of l_1 is true and of l_2 is false (so that $\neg l_2$ is true). The assignment α is a complete assignment if it assigns a boolean value to all the variables in V; otherwise it is a partial assignment.*

Notation 1.1 *Let ϕ be a CNF formula and l be a literal. We denote $\phi|_l$ the simplification of ϕ by the literal l. Formally, we have:*

$$\phi|_l = \{cl \setminus \{\neg l\} \mid cl \in \phi \text{ and } l \notin cl\}$$

We can extend this notation to any assignment α as follows:

$$\phi|_\alpha = \{cl \setminus \{l \in cl \mid \neg l \in \alpha\} \mid cl \in \phi \text{ and } cl \cap \alpha = \emptyset\}$$

A literal l is satisfied (falsified) by an assignment α if $l \in \alpha$ ($\neg l \in \alpha$). A clause cl is satisfied by the assignment α if at least one literal in it is satisfied by α. A formula ϕ is satisfied by α if all its clauses are satisfied by α. In such case, if α is a complete assignment, we say that α is a solution of ϕ. We also have some specific clauses defined below:

- **Unit (Binary) Clause**: clause with exactly one literal (two literals) inside.
- **Empty Clause**: clause contains no literal, which is denoted by $\square$ or $\emptyset$. Empty clause cannot be satisfied, so it is logically false.
- **Tautological Clause**: clause contains both literal l and $\neg l$. Tautological clause can be satisfied with any assignment of l, so it is logically true.

1.3 Conclusion

In this chapter, we introduced some essential concepts from complexity theory and propositional logic that are foundational for the remainder of this manuscript. We have reviewed key notions in computational complexity, such as time and space complexity, decision and optimization problems, and the fundamental complexity

classes P, NP, and EXP. Additionally, we explored propositional logic, the conjunctive normal form (CNF), and relevant logical constructs like literals and clauses. These concepts provide the necessary theoretical framework to understand and engage with the advanced topics we will discuss in the following chapters. In the next chapter, we will introduce the Satisfiability (SAT) problem, which is a fundamental decision problem that lies at the core of both artificial intelligence and complexity theory.

2 Satisfiability

Sommaire

2.1 Definition and Variants . . . 24
2.2 Complete Algorithms . . . 26
2.2.1 The DP Algorithm . . . 26
2.2.1.1 Resolution Rule . . . 26
2.2.1.2 The DP Algorithm . . . 27
2.2.2 The DPLL Algorithm . . . 29
2.2.3 The CDCL Algorithm . . . 35
2.2.3.1 Clause Learning and Non-Chronological Backtracking 35
2.2.3.2 Variable Branch Heuristic . . . 39
2.2.3.3 CDCL Algorithm . . . 42
2.3 Incomplete Algorithm . . . 42
2.4 Simplification Algorithms . . . 46
2.4.1 Bounded Variable Elimination . . . 46
2.4.2 Failed, Implied, Equivalent Literals Detection . . . 50
2.4.3 Clause Garbage Collection . . . 50
2.5 Conclusion . . . 51

In this chapter, we introduce the definition of satisfiability (SAT) problem and some of its variants in Section 2.1. The algorithms to solve it can be divided into two groups, the complete algorithms prove the soundness of unsatisfiable problems, while the incomplete algorithms usually find the solution for satisfiable instances faster, but it cannot prove the unsatisfiability. We will introduce the two classes separately in Section 2.2 and Section 2.3. Besides them, in Section 2.4, there are some simplification approaches in SAT solvers which aim to simplify the formula and reduce the difficulty of solving. Finally, we do a short conclusion.

2.1 Definition and Variants

Given a Conjunctive Normal Form (CNF) formula ϕ, the propositional satisfiability problem (SAT) aims to find an assignment satisfying all clauses in ϕ, or prove that such an assignment does not exist. In the former case, ϕ is said to be *satisfiable*, and in the latter case, ϕ is said to be *unsatisfiable*. SAT was the first problem proven to be NP-complete [58], and many other classical problems in the literature were later

shown NP-complete by reduction to SAT, such as Clique, Set Covering, and Travelling Salesman Problem (TSP) [88].

As a basic problem in logic, the SAT problem provides a bridge between mathematics and logic, and even information science [164]. Meanwhile, it is also the key to studying the completeness and decidability of logical systems, and the foundation of computational complexity theory[58]. For example, Heule et al. promoted Boolean Pythagorean Triples problem [82], Schur Numbe Five problem [81], Keller's conjecture [32] and other problems with SAT. Bright et al. used SAT solvers to prove that Lam's problem does not exist in geometry [33].

Beyond theoretical interest, SAT also has a wide range of practical applications due to its expressive power and close relation to many combinatorial and decision-making problems. With advances in SAT solvers, many real-world problems can now be efficiently tackled. For example in the field of Electronic Design Automation (EDA) [66], SAT solvers are used in the Equivalence Checking [92, 93] to verify that two representations of a circuit are functionally equivalent, and in Model Checking [25] to ensure that a circuit adheres to certain properties such as safety and liveness. In Software Verification that aims to ensure correctness or quality of computer software, SAT solvers are used to detect bugs and verify the safety of softwares [66]. Besides, SAT solvers are also used in Job-Shop Scheduling Problems (JSSP) [89], Hamiltonian Cycle Problem [184] and Cryptographic [176] et al.

There are many variants of the SAT problem. Some take special formulas as input: k-SAT restricts the size of the clauses in the input CNF formula to at most k-literals, Horn-SAT considers formulas where each clause contains at most one positive literal. The others have Quantified Satisfiability (QSAT), Satisfiability Modulo Theories (SMT) and Sharp Satisfiability (#SAT). The reader can refer to the Handbook of Satisfiability [19] for further details on these classical problems as they are out of the scope of this thesis. A particular extension we will study however is Maximum Satisfiability problem (MaxSAT), the natural optimization extension of SAT which will be introduced in Chapter 3.

A SAT solver is a software used to determine whether a given SAT formula ϕ can be satisfied. Its input is ϕ in a standard format (usually it's DIMACS format [1]), and its output is "SAT" with an assignment (i.e., the solution) if ϕ is satisfied, otherwise it outputs "UNSAT". Nowadays, the most widely used SAT solvers include: MiniSAT [67], Glucose [166], Cadical [23], Kissat [26] et al.

Aims to evaluate and improve SAT solvers, the SAT competition is held annually since 2002. All submitted solvers are tested on the provided experimental platform with the same set of benchmarks and time limit. The benchmarks include real-world, crafted and random instances and will be published after the competition to prevent targeted tuning. The solvers are ranked by their PAR2 scores [2]. The results of the

[1] https://web.archive.org/web/20190325181937/https://www.satcompetition.org/2009/format-benchmarks2009.html

[2] Penalized Average Runtime with a factor of 2. If a SAT solver successfully solves an instance within the time limit, the actual runtime is recorded. Otherwise, the runtime for that instance is penalized by assigning it a runtime equal to twice the time limit. PAR2 socre is the average of the runtime of all

SAT competitions show the efficiency of the solvers, and the techniques used by the winning solver are often adopted by other solvers.

Over the years, significant progress has been made in the development of SAT solvers, driven by both theoretical insights and practical demands. The performance of SAT solvers has dramatically improved, enabling the resolution of large, complex instances that were previously intractable. This improvement stems from a variety of algorithmic innovations, including complete and incomplete algorithms, and simplification techniques. In the following sections, we will present these core algorithms and methods that power modern SAT solvers, illustrating how they contribute to the success of SAT solving.

2.2 Complete Algorithms

In this section, we present major complete algorithms for SAT solving in the literature. The complete algorithms play a crucial role in guaranteeing a solution, i.e., to find a satisfying assignment or prove that no such assignment exists. The first complete algorithm in SAT solving is Davis-Putnam (DP) algorithm proposed by Martin Davis and Hilary Putnam in 1960 [62]. The number of clauses grows rapidly in DP, thus the memory consumption, which limits its usage in large SAT instances. After two years, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm was introduced [61]. It searches the solution by constructing a binary assignment tree, which avoids the growth of memory, but it is still not efficient. Nowadays, the algorithm widely used in modern SAT solvers is Conflict Driven Clause Learning (CDCL) algorithm [145]. It is based on DPLL, but equipped with many techniques that significantly improve its performance.

2.2.1 The DP Algorithm

2.2.1.1 Resolution Rule

To introduce DP algorithm we should first introduce the *resolution rule* for clauses in sat solving. And it can be easily generalized to the resolution rule for sets of clauses, as shown in the following:

Definition 2.1 (Resolution rule for Clauses) *Given two clauses* $cl = \{x, a_1, ..., a_n\}$ *and* $cl' = \{\neg x, b_1, ..., b_m\}$ *in* ϕ*, the resolution rule, by* resolving cl and cl' on x*, derives a new clause* $cl \otimes_x cl' = \{a_1, ..., a_n, b_1, ..., b_m\}$*, called the* resolvent *of* cl *and* cl'*. Variable* x *is called the* pivot *in this application of resolution. The resolution rule is usually shown as follows:*

$$\frac{cl = \{x, a_1, ..., a_n\} \qquad cl' = \{\neg x, b_1, ..., b_m\}}{cl \otimes_x cl' = \{a_1, ..., a_n, b_1, ..., b_m\}}$$

instances, i.e., PAR2= $\frac{\sum_{i=1}^{n} T_i}{n}$, in which T_i is the runtime for instance i, and n is the total number of instances.

Definition 2.2 (Resolution rule for Clauses Sets) *Given a CNF formula ϕ, ϕ_x ($\phi_{\neg x}$) denotes the set of all clauses in ϕ containing literal x ($\neg x$). The resolution rule can be lifted to ϕ_x and $\phi_{\neg x}$ to derive the set of clauses $\phi_x \bigotimes_x \phi_{\neg x} = \{cl_1 \bigotimes_x cl_2 \mid cl_1 \in \phi_x, cl_2 \in \phi_{\neg x}, cl_1 \bigotimes_x cl_2 \; is \; not \; a \; tautology\}$, called the resolvent set of ϕ_x and $\phi_{\neg x}$.*

It is well-known that ϕ is unsatisfiable if and only if there is a sequence of clauses $cl_1, cl_2, \ldots, cl_k$, such that cl_k is the empty clause and each cl_i $(1 \leq i \leq k)$ is a resolvent of two clauses in $\phi \cup \{cl_1, \ldots, cl_{i-1}\}$ [154]. Given such a sequence of clauses, we can construct a binary resolution tree as follows: (1) every tree node is a clause in $\phi \cup \{cl_1, \ldots, cl_k\}$; (2) the root is cl_k; (3) every non-leaf node cl_i has two children in $\phi \cup \{cl_1, \ldots, cl_{i-1}\}$ whose resolvent is cl_i; and (4) every leaf node is a clause in ϕ. The resolution is said to be *regular* if no variable is resolved on more than once in any path from the root to a leaf in the resolution tree, and is *unrestricted* if there is no limitation for resolution. As seen later, both resolution methods are crucial in SAT solving. Specifically, regular resolution forms the basis of the DP algorithm, while the CDCL algorithm is closely related to unrestricted resolution.

2.2.1.2 The DP Algorithm

The DP Algorithm, which is the first complete algorithm in solving SAT, is based on regular resolution. As shown in Algorithm 1, given a SAT formula ϕ, the DP algorithm returns SAT if it's satisfiable, otherwise UNSAT. The set of V is the set of all variables in ϕ, and function $pickVariable(V)$ returns a variable x that can be resolved on, i.e., both x and $\neg x$ appear in clauses in ϕ (line 7). The DP approach iteratively removes variable x from ϕ by adding all resolvents of x into ϕ and deleting all clauses including x or $\neg x$ (lines 10-12). This process ends when ϕ becomes empty or no more resolution can be executed, in which case ϕ is satisfiable and we get the solution by simply assigning value *true* to all rest literals (lines 3-4, 8-9), or there is an empty clause that cannot be satisfied in ϕ, in which case the formula is unsatisfiable (lines 5-6). The variables removed during this process are also called *eliminated variables*.

In Example 2.1 we show the execution of DP algorithm on a SAT formula includes 6 variables and 9 clauses. This process can be presented as a binary tree following the construction rule in Section 2.2.1.1, as in Figure 2.1, where each node is a set of clauses including a special variable as the pivot. The two children nodes are clauses sets of literal x and $\neg x$ individually. The parent node is not exactly the resolvent of its children, to make the algorithm more clear.

Example 2.1 *Given a formula $\phi = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}, \{\neg x_1, \neg x_3\}, \{\neg x_1, \neg x_5\}, \{\neg x_3, \neg x_5\}, \{\neg x_2, \neg x_4\}, \{\neg x_2, \neg x_6\}, \{\neg x_4, \neg x_6\}\}$, we solve it by doing regular resolutions on variables $x_1, \ldots, x_6$ sequentially:*

1. *By resolving on x_1, we get clauses $\{\{x_1, x_2\}\} \bigotimes_{x_1} \{\{\neg x_1, \neg x_3\}, \{\neg x_1, \neg x_5\}\} = \{\{x_2, \neg x_3\}, \{x_2, \neg x_5\}$, and clauses $\{x_1, x_2\}, \{\neg x_1, \neg x_3\}, \{\neg x_1, \neg x_5\}$ are removed from ϕ, so that x_1 is eliminated.*

Algorithm 1: DP(ϕ), the SAT algorithm based on regular resolution

Input: ϕ: a CNF SAT formula
Output: SAT or $UNSAT$: the satisfiability of ϕ

```
1  begin
2      while true do
3          if φ = ∅ then
4              return SAT;
5          if □ ∈ φ then
6              return UNSAT;
7          x ← pickVariable(V);
8          if x = ∅ then
9              return SAT;
10         do regular resolution on φ_x and φ_¬x;
11         φ = {φ \ {φ_x ∪ φ_¬x}} ∪ φ_x ⊗_x φ_¬x;
12         V = V \ {x};
```

2. *By resolving on x_2, we get clauses $\{\neg x_3, \neg x_4\}, \{\neg x_3, \neg x_6\}, \{\neg x_4, \neg x_5\}, \{\neg x_5, \neg x_6\}$, and $\{x_2, \neg x_3\}, \{x_2, \neg x_5\}, \{\neg x_2, \neg x_4\}, \{\neg x_2, \neg x_6\}$ and x_2 are removed.*

3. *By resolving on x_3, we get $\{x_4, \neg x_4\}, \{x_4, \neg x_5\}, \{x_4, \neg x_6\}, \{\neg x_5, \neg x_6\}$, and $\{x_3, x_4\}$, $\{\neg x_3, \neg x_4\}, \{\neg x_3, \neg x_6\}, \{\neg x_3, \neg x_5\}$ and x_3 are removed.*

4. *By resolving on x_4, we get $\{\neg x_5, \neg x_6\}, \{\neg x_5\}, \{\neg x_6\}$, and $\{x_4, \neg x_5\}, \{x_4, \neg x_6\}$, $\{\neg x_4, \neg x_5\}, \{\neg x_4, \neg x_6\}$ and x_4 are removed.*

5. *By resolving on x_5, we get $\{x_6, \neg x_6\}, \{x_6\}$, and $\{\neg x_5, \neg x_6\}, \{\neg x_5\}, \{x_5, x_6\}$ and x_5 are removed.*

6. *There are two unit clauses $\{x_6\}$ and $\{\neg x_6\}$. By resolving on x_6 we get an empty clause $\square$ which cannot be satisfied, so that we prove that the formula ϕ is unsatisfiable.*

Note that in step 3, the first clause we get is $\{x_4, \neg x_4\}$, which is a tautology clause that is always satisfiable, so that we can delete it from the formula directly. The same happens to the first clause $\{x_6, \neg x_6\}$ in step 5.

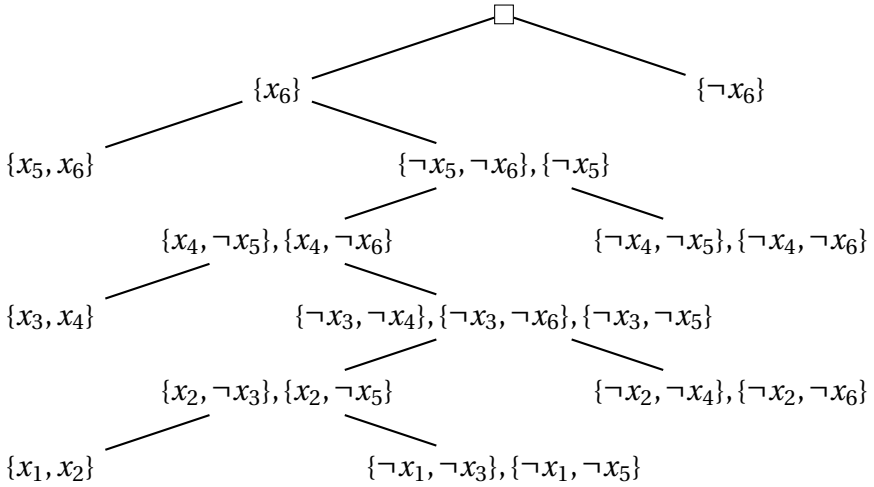


Figure 2.1: The solving process of DP algorithm

For a variable x whose literal x occurs in M clauses and $\neg x$ in N clauses, the regular resolution on it produces $M \times N$ new clauses in the worst case. As the step 2 in Example 2.1, we get 4 new clauses by resolving on x_2. It highlights the main issue of DP algorithm, that is, its high space complexity, as it tends to generate too many clauses. The huge memory request is also the reason that DP algorithm is no longer widely used nowadays. But, if we choose the pivot variables carefully, there won't have big memory increase and the formula can be simplified by removing variables and related clauses. For example in step 1, we removed 3 clauses with 2 clauses added; in step 3, we removed 4 clauses with 3 clauses added because of the tautology clause; and 4 removed with 3 added because of a duplicate clause in step 4. So in modern SAT solvers, regular resolution is usually used to simplify the formula, this approach is called bounded variable elimination, which we will introduce in Section 2.4.

2.2.2 The DPLL Algorithm

As the response to the space requirements of the DP algorithm, the DPLL algorithm was proposed after two years [61]. It's a more practical approach that improves upon the earlier DP algorithm and went through more than thirty years of improvement until it was replaced by CDCL. In contrast to the DP, which constructs a binary resolution tree during the solving process and assigns values to variables only after the resolution is complete, the DPLL assigns variables during the search process, building a binary assignment tree as it progresses. Before introducing the DPLL algorithm, we first need to highlight some key concepts.

Definition 2.3 (Decision Variable) *A decision variable refers to a variable that is explicitly chosen by the solver to assign a truth value during the search process. This decision is not forced by any constraints or propagation but is selected as part of the search strategy to explore possible solutions. The decision variable is chosen based on some heuristic or strategy to guide the search process efficiently.*

Definition 2.4 (Decision Level) *A decision level dl corresponds to the number of decisions made by the solver so far in the search tree. Each time the solver chooses a decision variable x and assigns it a value without it being forced (i.e., not derived from unit propagation), the decision level increases by 1. For those variables forced to be assigned after x, i.e., they are assigned by unit propagation or pure literal elimination, they have the same decision level as x. Level 0 is the initial level, where no variables have been assigned values by decisions.*

Definition 2.5 (Conflict) *A conflict (also known as a contradiction) happens when a clause cannot be satisfied, i.e., all literals in this clause are assigned value false so that it becomes empty. We use $\square$ to present a conflict.*

Example 2.2 *For a SAT formula $\phi = \{\{x_1, x_2\}, \{x_3\}\}$, x_1 is selected as the first decision variable and assigned value true, its decision level is 1. Then, we choose x_3 as the second decision variable with decision level 2. It is assigned false, so that a conflict happens because clause $\{x_3\}$ becomes empty.*

To speed up the solving process, the DPLL algorithm also incorporates several techniques, including backtracking, unit propagation, pure literal elimination and variable branch heuristic, which makes it more efficient than the original regular resolution-based approach.

The DPLL algorithm enhances the efficiency of SAT solving through several techniques. *Variable branch heuristic* strategically selects the next variable to assign based on estimations of variable quality, optimizing the search process. *Backtracking* allows the algorithm to flip latest variable assignments in case of conflicts, enabling exploration of alternative solutions without starting from scratch. *Unit propagation* accelerates the process by immediately assigning values to variables when a clause becomes a unit clause, thus simplifying the formula and potentially leading to quicker resolutions. And *pure literal elimination* further streamlines the solving process by allowing the algorithm to assign values to literals that appear without their negation, reducing the complexity of the formula. Together, these techniques make the DPLL algorithm significantly more effective than the traditional resolution-based approaches. And they are also utilized in CDCL, as we will see in Section 2.2.3. Below, we provide a detailed explanation of each of them.

Variable Branch Heuristic The branch heuristic are used to guide the decision variable selection. For example, variables can be selected simply in lexicographic or random order. However, a good branching heuristic can greatly improve the performance of the solvers. We will introduce those heuristics used in practice later in Section 2.2.3.2.

Backtracking The DPLL algorithm assigns value true or false to a decision variable. If a conflict arises, it backtracks, which means it cancels all assignments under current decision level dl, and try the opposite assignment of the decision variable. If the

second attempt also deduces a conflict, the search backtracks to the previous level, i.e., cancels the assignments until level $dl-1$ (included), and flip the assignment of the decision variable at level $dl-1$. This avoids exhaustively searching through all possible truth assignments. Since the algorithm simply undoes the most recent decision, we call it *Chronological Backtracking*, whereas the opposite is Non-Chronological Backtracking, which is used in the CDCL algorithm and will be introduced later. In the following we use the same formula as in Example 2.1 to show the process of assignment and backtracking.

Example 2.3 *We use the same SAT formula* $\phi = \{(x_1, x_2), (x_3, x_4), (x_5, x_6), (\neg x_1, \neg x_3), (\neg x_1, \neg x_5), (\neg x_3, \neg x_5), (\neg x_2, \neg x_4), (\neg x_2, \neg x_6), (\neg x_4, \neg x_6)\}$ *as in the example 2.1. The assignment of variables follows their numerical order, and for each variable, we first assign value true to it, then false. Figure 2.2 shows the process.*

1. *The first decision variable is* x_1*, by assigning true to* x_1 *at decision level 1, we get the simplified formula under* x_1*,* $\phi|_{x_1} = \{(x_3, x_4), (x_5, x_6), (\neg x_3), (\neg x_5), (\neg x_3, \neg x_5), (\neg x_2, \neg x_4), (\neg x_2, \neg x_6), (\neg x_4, \neg x_6)\}$*, and* $dl = 1$*.*

2. *The second decision variable is* x_2*, by assigning true to* x_2*, we get* $\phi|_{x_1 x_2} = \{(x_3, x_4), (x_5, x_6), (\neg x_3), (\neg x_5), (\neg x_3, \neg x_5), (\neg x_4), (\neg x_6), (\neg x_4, \neg x_6)\}$*, and* $dl = 2$*.*

3. *The third decision variable is* x_3*, by assigning true to* x_3*, we get* $\phi|_{x_1 x_2 x_3} = \{(x_5, x_6), \square, (\neg x_5), (\neg x_4), (\neg x_6), (\neg x_4, \neg x_6)\}$*, and* $dl = 3$*. There is an empty clause (conflict) in the formula, the algorithm cancels the assignment of* x_3 *and backtracks to level 2. Next, it will flip the assignment of* x_3*.*

4. *By assigning false to* x_3*, we get* $\phi|_{x_1 x_2 \neg x_3} = \{(x_4), (x_5, x_6), (\neg x_5), (\neg x_4), (\neg x_6), (\neg x_4, \neg x_6)\}$*, and* $dl = 3$*. No conflict happens under current assignment, the search continues.*

5. *By assigning true to* x_4 *at* $dl = 4$*, we get* $\phi|_{x_1 x_2 \neg x_3 x_4}$ *including an empty clause, so the assignment of* x_4 *is canceled and the algorithm backtracks to* $dl = 3$*.*

6. *By assigning false to* x_4 *at* $dl = 4$*, we get* $\phi|_{x_1 x_2 \neg x_3 \neg x_4}$ *that also include a conflict. By now we have tried the two assignment of* x_4 *and both cause an empty clause, and we have also tried the two assignments of* x_3*. So we should backtrack to level 2 and flip the assignment of* x_2*.*

7. *By assigning false to* x_2*, we get* $\phi|_{x_1 \neg x_2}$ *that includes no conflict, and the search continues from the current assignment.*

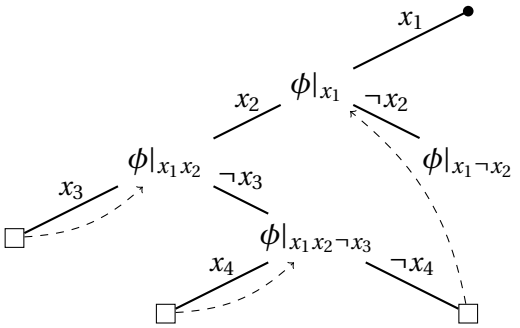


Figure 2.2: The process of backtracking, in which each branch represents an assignment to a variable (shown by x or $\neg x$ near the edge, and each node represents the formula simplified by the assignment. If there is a conflict in the simplified formula, we emphasize it by $\square$. The branches are represented by solid lines, and the backtracking paths are represented by the dashed lines. The arrow indicates the direction of backtracking.

Unit Propagation When a unit clause (e.g., x_1) is found, the algorithm must assign x_1 the value true to satisfy this clause. After this assignment, the formula is simplified by removing clauses containing the satisfied literal x_1, and removing falsified literal $\neg x_1$ from the remaining clauses containing it. If this simplification deduces new unit clauses, they are satisfied and the propagation continues until no more unit clauses rest or a conflict happens.

Example 2.4 *For a SAT formula $\phi = \{(x_1, \neg x_2), (x_2), (\neg x_1), (\neg x_3)\}$. We first satisfy unit clause (x_2) by assigning $x_2 = true$, then (x_2) is removed from ϕ, and $\neg x_2$ is removed from the first clause. We now have a new unit clause x_1 and it must be satisfied, so x_1 is assigned true. Then, $\neg x_1$ is removed from the third clause and it becomes empty. A conflict happens, and the unit propagation stops.*

Pure Literal Elimination Pure literals are literals that appear in only one polarity, they can be satisfied without any risk of conflict, and all clauses containing them are also satisfied, thus the formula is simplified. In the example above, $\neg x_3$ is a pure literal, we satisfy it by assigning false to x_3 and remove clause $(\neg x_3)$ from the formula.

The recursive version of DPLL algorithm is shown below. At each decision level, the algorithm applies pure literal elimination and unit propagation to simplify the formula (lines 6-7). Then, a decision variable is selected by branch heuristic and assigned (lines 8-9). If the assignment deduces a conflict, the algorithm backtracks to the latest assigned literal to branch on its negation (lines 10-14). If the formula becomes empty, it is satisfiable and the algorithm returns SAT; if it includes an empty clause, it is unsatisfiable and the algorithm returns UNSAT (lines 2-5). To prove the satisfiability of a given formula, the DPLL algorithm only needs to find one solution for it. But to prove the unsatisfiability, it should check all branches in this binary tree and prove that all assignments branches will produce an empty clause. In example 2.5 we show the execution of DPLL algorithm.

Algorithm 2: $DPLL(\phi, \alpha)$, a search algorithm

Input: ϕ: a CNF SAT formula, α: the assignment, $\alpha = \emptyset$ in the beginning
Output: SAT or $UNSAT$: the satisfiability of ϕ

```
1  begin
2      if φ = ∅ then
3          return SAT;
4      if □ ∈ φ then
5          return UNSAT;
6      φ, α ← PureLiteralElimination(φ, α);
7      φ, α ← UnitPropagate(φ, α);
8      l ← BranchHeuristic(φ);
9      α ← α ∪ {l};
10     if DPLL(φ, α) returns SAT then
11         return SAT;
12     else
13         α ← α \ {l} ∪ {¬l};
14         return DPLL(φ, α);
```

Example 2.5 *We consider the satisfiable formula* $\phi = \{cl_1, cl_2, cl_3, cl_4, cl_5\}$ *where* $cl_1 = (x_1, \neg x_2)$, $cl_2 = (\neg x_2, x_3)$, $cl_3 = (\neg x_3, x_4)$, $cl_4 = (\neg x_3, \neg x_4)$ *and* $cl_5 = (\neg x_1, x_2, x_3)$. *The search tree in Figure 2.3 depicts the execution of the DPLL algorithm on* ϕ *with lexicographic branching. The search backtracks each time a conflict happens, and it finally finds the assignment that makes* $\phi = \emptyset$, *thus satisfies the formula. The solution of* ϕ *is* $(0, 0, 0, 1)$.

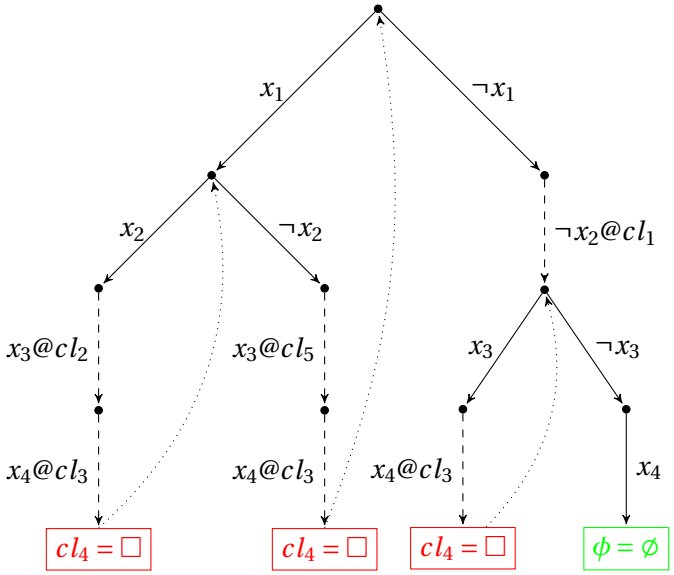


Figure 2.3: The search tree depicts the execution of the DPLL algorithm, where branching is made in lexicographic order of the variables. Solid, dashed and dotted arrows represent respectively branching, propagation and backtracking. $l@cl$ denotes that the clause cl is the reason of the propagation of literal l.

Since it only needs space to store the original formula, and no new clauses are created during search, DPLL avoids the huge memory cost in DP. But it also has some limitations listed below. Decades later, the CDCL algorithm was proposed and has now become a common algorithm in almost all SAT solvers, since it solves these limitations well. We will illustrate it in Section 2.2.3.

- When a conflict happens, the solver always flip the assignment of the latest decision variable, which may not helpful in avoiding the conflict. In Example 2.6 we illustrate it.
- The DPLL algorithm cannot avoid the same conflicts during the search process. It's clear in Figure 2.2, the reasons of the conflict in path $x_1x_2x_3$ and $x_1\neg x_2x_3$ are the same, that is variables x_1 and x_3 cannot be satisfied at the same time, as there is a clause $(\neg x_1, \neg x_3)$. But we can't avoid these repeated conflicts in DPLL search.

Example 2.6 *We have a SAT formula* $\phi = \{(\neg x_1, \neg x_4), (\neg x_1, x_4), (x_2, x_3), (x_3, x_5)\}$. *After assigning true to variables* x_1, x_2, x_3, *we get* $\phi|_{x_1x_2x_3} = \{(\neg x_4), (x_4)\}$. *Next, both true and false assignment to* x_4 *can produce a conflict. Following the rule in DPLL, we should flip the value of* x_3, *and then* x_2. *But there is no relation between the conflict and variables* x_2, x_3, *the reason of conflict is actually the value of* x_1 *is true. But DPLL cannot deal with this situation and thus waste a lot of time in useless search space.*

2.2.3 The CDCL Algorithm

The Conflict Driven Clause Learning (CDCL) algorithm effectively addresses these shortcomings of DPLL by learning from conflicts. It was first proposed in [140, 165] and quickly became the main technique in the solvers today due to its significant effectiveness. The solvers equipped with CDCL include: GRASP [165], (z)Chaff [144, 178, 136], MiniSAT [67], Glucose [166], Lingeling [21], MapleSAT solvers [127, 129, 130], Cadical [23] and Kissat [26]. The improvements of CDCL over DPLL include clause learning, non-chronological backtracking, and more efficient variable branch heuristics.

2.2.3.1 Clause Learning and Non-Chronological Backtracking

During the CDCL search, the relationship between decision variables and forcibly assigned variables (i.e., assigned by unit propagation) can be presented by the implication graph. If the algorithm finds a conflict, by constructing the implication graph, we get all the variables and clauses involved in the current conflict. When a conflict happens, by analyzing the implication graph, we learn the reason of this conflict. The analyzing process is called conflict analysis, and the knowledge we learn is represented by a clause, which we call learnt clause. The learnt clauses are added into the formula, to prevent the same conflicts from happening in the future search, i.e., pruning the search space.

Definition 2.6 (Implication Graph) *An implication graph G is a directed labeled acyclic graph where each node represents an assignment $l@dl$, in which l is a literal assigned to true and dl is its decision level. The edges represent the reasoning relationship between variables, on which the clause used in the reasoning are marked. In the following we show an example of implication graph. For nodes without incoming edges, they are decision variables. For nodes without outcoming arrows, the unit propagation stops there. For any node $l@dl$ with incoming edges, the combination of the negation of all its incoming nodes' literals represent the reason (clause) of its assignment.*

Example 2.7 *For the SAT formula $\phi = \{cl_1, cl_2, cl_3, cl_4, cl_5, cl_6\}$, in which $cl_1 = (\neg x_1, \neg x_7, \neg x_9)$, $cl_2 = (x_2, \neg x_4, x_5)$, $cl_3 = (\neg x_4, \neg x_6)$, $cl_4 = (\neg x_5, x_6, x_7)$, $cl_5 = (\neg x_7, x_8)$, $cl_6 = (\neg x_8, x_9)$. The variables x_1, x_2, x_3 and x_4 are selected as the decision variable and assigned, as in Figure 2.4, by unit propagation we get a conflict at level 4, and an implication graph is constructed from it.*

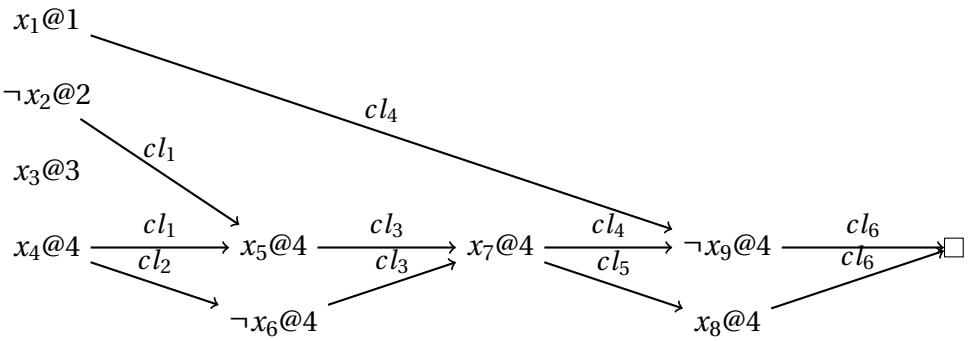


Figure 2.4: The implication graph of Example 2.7. The clause cl_6 becomes empty because unit propagation falsifies literal x_8 and $\neg x_9$.

In Figure 2.4, the conflict happens in decision level 4, which we call the *conflict level*. It's clear that at level 4, all paths from the decision variables to the conflict pass through two points $x_4@4$ and $x_7@4$. We call such points *unique implication point (UIP)*. It's a critical concept used during conflict analysis, because it determines the occurrence of conflict. Under the current decision level, if the assignment of UIPs are unchanged, the same conflict will occur inevitably.

Definition 2.7 (Unique Implication Point (UIP)) *The unique implication points are points that all paths from the decision variable to the conflict pass through under current conflict level. There may be more than one UIPs, the one closest to the conflict is First-UIP, and the one farthest from it is called Last-UIP. In Figure 2.4, $x_7@4$ is the First-UIP and $x_4@4$ is the Last-UIP.*

In order to obtain a learnt clause from the conflict, the SAT solvers will maintain a clause $learnt$ and initialize it to the empty clause encountered. Then the algorithm will follow the reverse order of variable assignments, and successively replace $learnt$ by the resolvent of itself and the corresponding clauses in the implication graph according to the variables appearing in $learnt$, until a UIP is encountered. At this time, the algorithm returns $learnt$ as a learnt clause. To make it more clear, we can draw a cut on the right side of UIP, so that UIP appears on the side close to the decision variable. At this time, the clause composed of the negation of the literals on the left side of the cut is the learnt clause. Different UIPs will generate different learnt clauses. For example in Figure 2.5 and Figure 2.6, with First-UIP $x_7@4$ we get learnt clause $learnt_1 = (\neg x_1, \neg x_7)$; however with Last-UIP $x_4@4$ we get clause $learnt_2 = (\neg x_1, x_2, \neg x_4)$. Different learnt clauses have different qualities, and it's generally believed that the learnt clauses corresponding to First-UIP have higher quality. Early CDCL solvers would generate learnt clauses at multiple UIPs, but modern CDCL solvers generally only generate one learnt clause at the First-UIP and exit the analysis process.

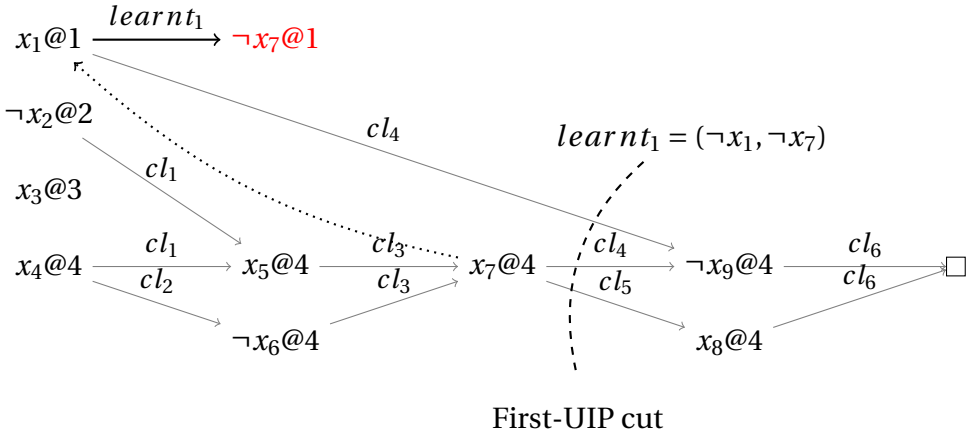


Figure 2.5: The implication graph with First-UIP cut. The cut is represented by the dashed curve edge between node $x_7@4$ and $x_9@4$. The literals at its left side are x_1 and x_7, so we get learnt clause $learnt_1 = (\neg x_1, \neg x_7)$. After adding $learnt_1$ into the formula, the solver backtracks to the second highest level in it, that is, the level of x_1. The backtracking is represented by a dotted curve arrow from $x_7@4$ to $x_1@1$. Since $learnt_1$ becomes a unit clause after the backtracking, $\neg x_7$ should be satisfied at level 1 by unit propagation, as shown by the arrow from $x_1@1$ to $\neg x_7@1$.

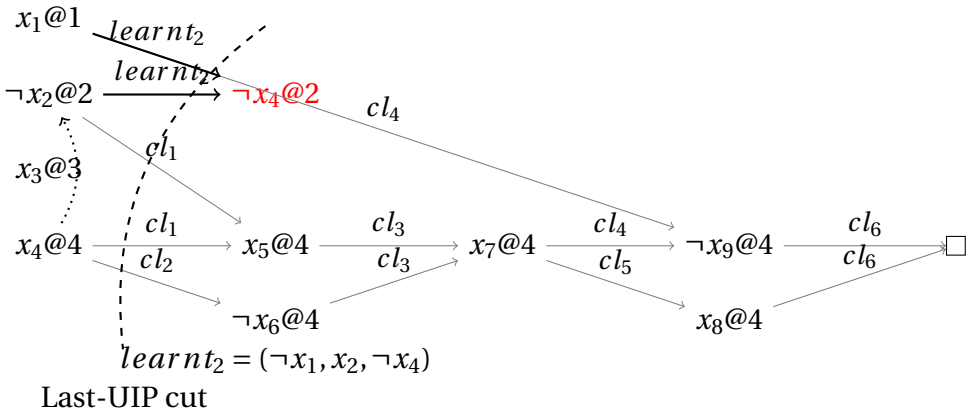


Figure 2.6: The implication graph with Last-UIP cut. The cut is represented by the dashed curve edge. The literals at its left side are $x_1, \neg x_2$ and x_4, so we get learnt clause $learnt_2 = (\neg x_1, x_2, \neg x_4)$. After adding $learnt_2$ into the formula, the solver backtracks to level of x_2. The backtracking is represented by a dotted curve arrow from $x_4@4$ to $\neg x_2@2$. Since $learnt_2$ becomes a unit clause after the backtracking, $\neg x_4$ should be satisfied at level 2 by unit propagation, as shown by the arrow from $x_1@1$ and $\neg x_2@2$ to $\neg x_4@2$.

Algorithm 3 depicts such a conflict analysis on the implication graph using First-UIP schema. The process to derive the learnt clause $learnt$ from the conflict is the

so-called conflict-driven clause learning (lines 2-5). The analysis stops at the point where $learnt$ contains only one literal (first UIP) from the conflict level, the algorithm then returns $learnt$ as the learnt clause and bl as the backtracking level(lines 6-7). The variables involved in the conflict analysis, including those appear in the conflict clause and those in the clauses leading to conflict, are crucial, as they are likely to deduce conflicts. By prioritizing these variables for future decisions, the algorithm can focus on the most critical parts of the search space, potentially resolving conflicts more efficiently and accelerating the solving process. We will introduce the details in Section 2.2.3.2. The learnt clause will be added to the formula to prevent the solver from revisiting the same conflict in the future. And the backtracking level determines how far the solver should backtrack, allowing it to undo decisions made at higher levels that contributed to the conflict.

Algorithm 3: Analyse($learnt$, d), conflict analysis using First-UIP schema

Input: $learnt$: an empty (or falsified) clause containing all its falsified literals, d: the level in which $learnt$ is falsified
Output: $learnt$: a falsified clause containing only one literal falsified in level d, bl: the backtracking level

```
1 begin
2     while learnt contains at least two literals in level d do
3         l ← the most recently assigned literal in learnt;
4         let r = {¬l, a_1, ..., a_k} be the unit clause falsifying l in the implication
            graph (a_1, ..., a_k are all falsified);
5         learnt ← resolving learnt and r on the variable of l;
6     bl ← the second highest level in learnt;
7     return learnt, bl;
```

The quality of learnt clauses are different. Normally, the shorter the clause length, the stronger the constraint, and the higher the possibility of pruning, so clauses with short length are considered good clauses. In 2009, Simon et al. proposed to use the number of decision levels involved in a learnt clause to measure its quality, and named it Literal Block Distance (LBD) [166]. Since it estimates clause quality more accurately than clause length, this criterion has also been widely adopted by subsequent SAT solvers. In the example above, the learnt clause generated by First-UIP is $learnt_1 = (\neg x_1, \neg x_7)$, with both length and LBD being 2; but the length and LBD of $learnt_2 = (\neg x_1, x_2, \neg x_4)$ are 3. The difference in clause length and LBD also explains the effectiveness of First-UIP schema, as the clauses generated by First-UIP are generally shorter in length and have smaller LBD.

Definition 2.8 (Literal Block Distance (LBD)) *LBD is a heuristic used in CDCL SAT solvers to measure the quality of learnt clauses. It computers the number of decision levels (or "blocks") involved in a learnt clause. Specifically, it counts how many different decision levels the literals of a clause come from. A lower LBD learnt clause indicates*

that the literals in it are decided in fewer distinct decision levels, thus it's critical for guiding the search process.

Generally speaking, the backtracking is divided into chronological backtracking and non-chronological backtracking. Chronological backtracking refers to backtracking to the position of the most recent decision and flipping its assignment, or returning to the upper decision level, and will not backtrack across multiple levels, as we introduced in section 2.2.2. The backtracking here is non-chronological backtracking, which refers to backtracking across multiple levels, generally backtracking to the second highest level in the learnt clause obtained by conflict analysis. Since the learnt clause includes UIP, it has only one variable at the conflict level, after backtracking, the learnt clause becomes a unit clause, and the assignment of UIP can be inverted. Figures 2.5 and 2.6 respectively show non-chronological backtracking that depend on First-UIP and Last-UIP. And we present chronological backtracking in Figure 2.2.

2.2.3.2 Variable Branch Heuristic

The variable branch heuristic selects the next decision variable. From practice, the quality of decision variables decides the performance of CDCL solvers. The early branch heuristic methods are based on the information from clauses, for example, the Maximum Occurences on Minimum Size Clause (MOMS) approach prefers literals that appear most frequently in the shortest clause. This information is static and is not affected by the solving process. However, the most common variable branch heuristic techniques nowadays are all dynamic: each variable has a score that is continuously updated throughout the solving process and reflects the variable's potential to cause a conflict. The representative methods include: Variable State Independent Decaying Sum (VSIDS), Condlict History-Based (CHB), Learning Rate Branching (LRB) and Variable Move-To-Front (VMTF). The detailed introduction are below.

Variable State Independent Decaying Sum (VSIDS) [145] This method was first introduced more than 20 years ago in Chaff solver, and still a key approach in SAT solving nowadays. In VSIDS, each variable maintains an *activity*, which is a score that initialized to 0 at the beginning of the search and updated after each conflict. When choosing the next branching variable, the one with the highest activities will be picked and assigned a value, then the unit propagation is executed. The purpose of VSIDS is to select variables that have appeared frequently in recent conflicts. Two operations are periodically performed on variables activity to achieve this in Chaff:

- **Bump:** after each conflict, the activity of the variables in the learnt clause is incremented by 1.

$$activity(v) = activity(v) + 1, \forall v \in learnt$$

- **Decay:** all activities are multiplied by a decay factor d ($0 < d < 1$) periodically, i.e., after m conflicts [3].

$$activity(v) = activity(v) * d, \forall v \in V$$

, V is the set of variables in ϕ

The VSIDS method in MiniSat [67] has some modifications. The activities of the variables in all the clauses involved in conflict analysis (we represent the clauses set by C) are bumped. Furthermore, the decay is performed at each conflict ($m = 1$) and the decay factor $d = 0.95$ in MiniSat. To save time, the decay is achieved by bumping the activities of variable with bump factor b. As in Chaff, b is initialized to 1, and it is increased by $1/d$ after each conflict, until reaching a certain limit [4], after which the bump factor b and the activities of all variables are scaled down.

$$activity(v) = activity(v) + b, \forall v \in cl, cl \in C$$

$$b = b * (1/d)$$

The Normalised VSIDS (NVIDS) variant, introduced in [20] and implemented in the Lingeling solver [21], represents an exponential moving average method. The activity points out how often a variable occurred in set C. To this end, given the decay factor d, after each conflict, the activity of all the variables are decayed, including those involved in the conflict. The activities are bumped (and decayed) as follows.

$$activity(v) = d * activity(v) + (1 - d)$$

NVIDS updated variables activities after each conflict, which is time-consuming, so the Exponential VSIDS (EVSIDS) was also introduced in [20]. It is inspired by MiniSAT's sclaing and maintains an exponential increasing score increment $g^{i}(v)$ where $g = 1/d$ and $i(v)$ denotes the conflict index of variable v, i.e., the number of conflicts when a variable was last bumped. Variable activities and the score increment can also be rescored occasionally. As such, only activities of variables that are involved in the conflict need to be updated as follows:

$$activity(v) = activity(v) + g^{i(v)}$$

Finally, the Average Conflict-Index Decision (ACIDS) variant introduced in [28] ensures that the influence of earlier conflicts decreases exponentially with respect to the bumping steps. Given a variable $v \in V$, the activity is updated as follows:

$$activity(v) = (activity(v) + i(v))/2$$

[3] $d = 0.5$ and $m = 256$

[4] the floating point number limit is used in MiniSat

Conflict History-Based (CHB) Heuristic [127] This heuristic, implemented in the MapleCOMSPS solver [130], is based on the Exponential Recency Weighted Average (ERWA) [169] and favors the variables involved in recent conflicts as in VSIDS. CHB maintains a score (or activity) for each variable $v \in V$, initially set to 0. The score is updated when a variable is branched on, propagated, or asserted using ERWA as follows:

$$score(v) = (1-\alpha) \times score(v) + \alpha \times r(v)$$

The parameter $0 < \alpha < 1$ is the step-size, initially set to 0.4 and decayed by 10^{-6} after every conflict to a minimum of 0.06. $r(v)$ is the reward value for variable v which can decrease or increase the likelihood of picking v. Higher rewards are given to variables involved in recent conflicts according to the following formula:

$$r(v) = \frac{multiplier}{Conflicts - lastConflict(v) + 1}$$

where $Conflicts$ denotes the number of conflicts that occurred since the beginning of the search and $lastConflict(v)$ is updated to the current value of $Conflicts$ whenever v is present in the clauses involved in conflict analysis. $multiplier$ is set to 1.0 when branching, propagating or asserting the variable that triggered the score update lead to a conflict, otherwise it is set to 0.9. The idea is to give extra rewards for variables producing a conflict.

Learning Rate Branching (LRB) [129] A recent variant of CHB called the Learning Rate Branching (LRB) heuristic was introduced in [129]. This variant uses a more refined reward function which favors the variables that contributed the most in recent conflicts. Whenever a variable is unassigned through backtracking its score is updated using ERWA as in CHB but using a different reward function defined as follows:

$$r(v) = \frac{nbConflicts(v)}{age(v)}$$

where:

- $age(v)$ denotes the number of conflicts encountered in the period between the assignment of a value to v and its unassignment during backtrack.
- $nbConflicts(v)$ denotes the number of conflicts in which the variable v is involved during the same period.

This variant also included a decay component where the scores of unsigned variables are decayed by a factor $0 < f < 1$ after each conflict [5]. The intuition behind the LRB heuristic is that a free variable contributing to most conflicts during the period in

[5] f is set to 0.95 in [129] and the authors suggest performing the decays in batch, i.e., multiply the score of a variable a single time by f^k after k conflicts where it remains unassigned instead of performing k multiplications.

which it was assigned a value is likely to help derive new conflicts quickly if it is assigned again.

Variable Move-To-Front (VMFT) [155] This heuristic implemented in the Siege solver [6] simply moves a subset of the variables in each learned clause to the front of the decision queue. The size of this subset is a small constant m [7] and the variables are positioned arbitrarily at the front of the queue. This heuristic does not necessarily require associating scores to variable. However an efficient implementation is described in [28] where variables are simply scored by their conflict index.

2.2.3.3 CDCL Algorithm

There are four core components in the CDCL algorithm: the unit propagation performs Boolean logic reasoning, the conflict analysis accumulates knowledge from conflicts for pruning, the branching heuristic controls the order of search; and the backtracking cancels assignment until given level. Algorithm 4 describes the process of CDCL approach. It executes unit propagation at the top level according to the current assignment and records the conflict in $confl$ (line 3). If unit propagation leads to a conflict, the algorithm analyzes the reason of conflict and gets a learnt clause, it is added to the formula to prune search space in the future searching to avoid the solver stepping into the same conflict (line 4-6). Besides, a backtracking level bl is returned and used to guide backtracking. If the backtracking level is 0 or below, the conflict occurs at the decision level 0, that is, the original formula can deduce a conflict without any assignment, it's unsatisfiable (lines 7-9). If unit propagation cannot deduce the conflict, we pick and assign the next decision variable by branching heuristic. If the function BranchingHeuristic() returns $\emptyset$, all variables have been assigned without causing a conflict, so the formula is satisfiable and we find a solution. Otherwise, the branch function assigns an unassigned variable, and the unit propagation continues (lines 11-13).

2.3 Incomplete Algorithm

The incomplete algorithms of the SAT problem is mainly based on the Stochastic Local Search (SLS) technology, which abstracts the process of SAT solving into the optimization problem that aims to minimize the number of unsatisfied clauses. The optimization problem is solved when there are no unsatisfied clauses. However, SLS does not store the search history and cannot completely traverse the search space, thus it cannot prove the unsatisfiability of the problem. Incomplete algorithm and complete algorithm excel in different areas and have complementary solving capabilities. When the problem is large in scale and has no obvious structural features,

[6] https://www2.cs.sfu.ca/research/groups/CL/software/siege/
[7] $m = 8$ in [155]

Algorithm 4: CDCL(ϕ), the search algorithm in modern SAT solvers

Input: ϕ: a CNF SAT formula
Output: SAT or $UNSAT$: the satisfiability of ϕ

```
1  begin
2      while true do
3          confl ← UnitPropagate();
4          if confl ≠ ∅ then
5              bl, learnt ← Analyse(confl, d);
6              φ ← φ ∪ {learnt};
7              if bl < 0 then
8                  return UNSAT;
9              Backtracking(bl);
10         else
11             v ← BranchingHeuristic();
12             if v = ∅ then
13                 return SAT;
```

the search cost of CDCL is expensive, meanwhile, lightweight SLS method is usually one of the most effective methods [83].

To review the SLS method, some concepts must be defined.

Definition 2.9 (Search Space) *For a SAT formula ϕ with n variables, the search space of it is the set of all possible complete assignments. Thus, there are 2^n assignments in the search space of ϕ.*

Definition 2.10 (Candidate Solution) *A candidate solution α is an element, i.e., a complete assignment in the search space.*

Definition 2.11 (Neighborhood and Neighbor) *For a candidate solution α, the neighborhood of it is defined as its Hamming neighborhood, that is, the set of other candidate solutions that differ from α in only one variable assignment. Any element in the set is a neighbor of α.*

The SLS algorithm can start from any candidate solution α in the search space. By evaluating its neighbors, it then walk to a neighbor position that seems to be feasible, the process continues until a solution is found. A general SLS algorithm framework, which is also applicable to other problems besides SAT, is shown in Algorithm 5. To start, the SLS approach constructs a complete assignment α as the initial solution, usually by a greedy or random strategy, and initializes the scoring function and data structures (lines 2). Subsequently, the algorithm continuously selects a variable (line ***) based on the scoring function, flip it, and update the relevant scores and data structures (lines 7-9) until α is a solution, where the algorithm returns α (lines 4-5). If

the algorithm does not find a solution during the given time, it returns "UNKNOWN" (line 10).

Algorithm 5: SLS(ϕ), the general stochastic local search algorithm

Input: ϕ: a CNF SAT formula
Output: α: if the algorithm finds a solution, or $UNKNOWN$: otherwise

```
1  begin
2      α ← Initialization();
3      while the time limit is not met do
4          if ∀cl ∈ φ, cl is satisfied then
5              return α;
6          else
7              v ← VariableHeuristic();
8              Flip(v);
9              Update();
10     return UNKNOWN;
```

When applying SLS to SAT, there are four crucial components that play a key role in the algorithm's effectiveness. First, solution initialization generates an initial assignment, typically at random, to start the search process. Next, scoring mechanisms evaluate solutions, variables, and clauses to guide the search towards satisfying assignments. To handle potential cycling problems, techniques are employed to avoid repeatedly visiting the same solutions. And finally, variable heuristics guide the selection of variables for flipping, using strategies to maximize the likelihood of finding a satisfying solution efficiently. Together, these functions enable SLS to explore the search space in a focused and dynamic way.

Solution Initialization Most SAT solvers such as CCAnr [38], Sattime [100], ProbSAT [12], initialize the solution randomly. YalSAT [22] uses the local optimal solution got in the past to initialize at each time it restarts. Some use unit propagation to construct the initial solution [41]. In addition, there are also some hybrid algorithms that use CDCL solvers to construct the initial solution [9, 141], or use machine learning to generate a high-quality initial solution [180].

Scoring The scoring function includes the scoring of candidate solutions, variables and clauses, which are used to heuristically measure the difference between candidate solutions, the benefits brought by flipping variables and the importance of clauses.

We use $cost(\phi, \alpha)$ to present the score of candidate solution α in formula ϕ. Normally, $cost(\phi, \alpha)$ is the number of unsatisfied clauses under assignment α. If all neighbors of α have worse scores than it, the SLS search has encountered a *local optima*.

There are two different types of scoring functions $score(\phi, v)$ for variables. One is the timestamp of the variables' last change, which tends to select variables with earlier timestamps and encourage the solver to explore the search space [74]. The other one evaluates the quality change of the candidate solutions, the typical methods include:

- $score(\phi, v) = cost(\phi, \alpha) - cost(\phi, \alpha')$
  where α' is the candidate solution we get by flipping variable v in α [159].

- $score(\phi, v) = make(\phi, v) - break(\phi, v)$
  where $make(\phi, v)$ is the the number of clauses becoming satisfied from unsatisfied after flipping v, $break(\phi, v)$ is the number of clauses becoming unsatisfied from satisfied after flipping v [162].

The clauses scoring, also called clauses weighting, is a key technology in dynamic local search. We use $w(cl)$ to represent the weight of clause cl, representative methods include:

- **Smoothed Descent and Flood (SDF)**[157]: When SLS reaches local optima, SDF increases the weight of unsatisfied clauses and normalizes the clause weights so that the maximum cost difference between any two flips remains unchange. Then, the weights of all satisfied clauses are smoothed by the function $w(cl) = \rho w(cl) + (1-\rho)\overline{w_{sat}}$, where $\overline{w_{sat}}$ is the average of the weights of all satisfied clauses, and $0 < \rho < 1$.

- **Exponentiated Subgradient (ESG)**[158]: When the local optima is reached, with a fixed probability, ESG randomly flips a variable from all variables that appear in unsatisfied clauses; otherwise, the clauses weights are updated. When updating the weights, the satisfied clauses and unsatisfied clauses are multiplied by different coefficients, and then all weights are smoothed by function $w(cl) = \rho w(cl) + (1-\rho)\overline{w}$, where $\overline{w}$ is the average of all clause weights, and $0 < \rho < 1$.

- **Scaling and Probabilistic Smoothing (SAPS)**[85]: The same as ESG, when the local optima is reached, with a fixed probability, SAPS also randomly flips a variable from all variables that appear in unsatisfied clauses; otherwise, the clauses weights are updated. When updating the weights, the unsatisfied clauses are multiplied by a coefficients, and then their weights are smoothed with a certain probability through the function $w(cl) = \rho w(cl) + (1-\rho)\overline{w}$.

- **Pure Additive Weighting Scheme (PAWS)**[172]: The weights of all unsatisfied clauses will be incremented by 1 when a local optima is reached. When the number of clause weight increase reaches a threshold, the weights of all clauses will be reduced by 1. PAWS and its improved version have been applied to many solvers such as CCASat [37], EagleUP [73], and Sparrow2011 [100].

Cycling Problem SLS algorithm suffers from the cycling problem [143], i.e., revisiting a candidate solution that has been visited recently, which seriously affects the solving efficiency. To deal with it, there are two types of common methods. One is based on the tabu strategy, which prohibits the flipping of variables that have been flipped within the tabu step. The other is based on configuration checking [43], which prohibits the flipping of a variable when its neighbors have not changed or the clause state [131] has not changed.

Variable Heuristic The heuristic methods are used to guide the selection of flipping variable v, and the main strategies can be divided into two categories. One is pure random walk algorithms represented by WalkSAT [162], Novelty [142, 111, 99], and ProbSAT [12]. This type of algorithm randomly selects a variable from the unsatisfied clause to flip according to different probability distributions or strategies [162]. The other use a two-phases selection heuristics that comprehensively considers intensification and diversification. In the intensification mode, it greedily selects variables according to their score, while in the diversification mode, it uses a random walk mode to jump out of the local optima and update the clause weights. The representative solvers include GWSAT [160], Sparrow2011 [11], Sattime [100], and CCAnr [38].

2.4 Simplification Algorithms

The simplification algorithms in SAT are used to reduce the complexity of the problem and optimizing memory usage. They can be classified into two groups: Bounded variable elimination and failed/implied/equivalent literals detection remove variables from the formula, thus decrease the size of binary search tree. While learnt clauses elimination cleans useless learnt clauses periodically, ensuring enough space for new created learnt clauses. In this section, we will provide a detailed introduction to these simplification algorithms.

2.4.1 Bounded Variable Elimination

Bounded Variable Elimination (BVE) is based on regular resolution. It's actually the main technique of DP, which is introduced in section 2.2.1. It is first applied as a pre-processing technique in MiniSAT solver [67], recently it is used in state-of-the-art SAT solver Cadical [23, 24] and Kissat [26] as an in-processing interleaved with the CDCL search.

Although BVE is an important pre- and in-processing technique in state-of-the-art SAT solvers, it is quite special: While other formula simplification techniques such as clause vivification, self-subsuming resolution and equivalence reasoning usually produce a formula smaller and easier to solve, BVE can produce a formula harder to solve [151], depending on the subset of variables eliminated.

The state-of-the-art SAT solvers usually fix a variable ordering, which, together with some bounds on how much the formula is allowed to grow by eliminating a variable,

implicitly decides the subset of variables eliminated, because eliminating a variable can increase the number of clauses in which other variables occur, possibly preventing other variables from being eliminated subject to the bounds. In order to eliminate as many variables as possible without exceeding the bounds, usual variable orderings are based on the number of clauses in which a variable occurs positively or negatively, in such a way that the variable with the smallest number of occurrences is eliminated first and the number of occurrences of other variables grows more slowly.

The well-known DP procedure [62] solves a CNF formula ϕ by eliminating variables one by one: to eliminate a variable x from ϕ, remove $\phi_x \cup \phi_{\neg x}$ from ϕ and add the resolvent set $\phi_x \otimes_x \phi_{\neg x}$ to ϕ. The obtained formula is satisfiability equivalent to ϕ. Note that $\phi_x \otimes_x \phi_{\neg x}$ does not contain x, nor any tautology. The resolution in the DP procedure is regular because when it derives an empty clause, a regular resolution tree can be constructed by tracing back the history of the empty clause. It is shown in [153] that the DP procedure is very efficient for problems with low width, and that the advantages of DP and backtracking tree search can be combined using Bounded Variable Elimination (BVE). Inspired from this property, BVE was used as a pre-processing simplification technique in SAT solvers around twenty years ago [168].

Example 2.8 *Let* ϕ_x=$\{\{x, a_1, a_2\}, \{x, a_3\}, \{x, \neg b_1, \neg b_3\}\}$ *and* $\phi_{\neg x}$=$\{\{\neg x, b_1, b_2\}, \{\neg x, b_3\}\}$ *in* ϕ*, eliminating* x *consists in replacing* $\phi_x \cup \phi_{\neg x}$ *in* ϕ *by* $\phi_x \otimes_x \phi_{\neg x}$ = $\{\{a_1, a_2, b_1, b_2\}, \{a_1, a_2, b_3\}, \{a_3, b_1, b_2\}, \{a_3, b_3\}\}$.

Variable elimination, as a simplification technique in a SAT solver, should be used carefully for four reasons: (1) Eliminating a variable x from ϕ can increase the number of clauses in ϕ by up to $|\phi_x \otimes_x \phi_{\neg x}| - |\phi_x| - |\phi_{\neg x}|$. So, repeating this procedure without limit can result in an exponential increase in the number of clauses. (2) The resolvent of two clauses can be longer than the two clauses. (3) After x is eliminated, it cannot be resolved on anymore, hurting the power of clause learning in the sequel. (4) when a variable is eliminated in an in-processing, all learnt clauses containing it are lost.

Variable elimination implemented in state-of-the-art SAT solvers usually only considers the first two reasons, by setting a bound denoted *clauseNbGrowthLimit* on $|\phi_x \otimes_x \phi_{\neg x}| - |\phi_x| - |\phi_{\neg x}|$ and/or a bound denoted *resolventLengthLimit* for the produced resolvents: x is not eliminated if $|\phi_x \otimes_x \phi_{\neg x}| - |\phi_x| - |\phi_{\neg x}| > clauseNbGrowthLimit$ or any resolvent in $\phi_x \otimes_x \phi_{\neg x}$ contains more than *resolventLengthLimit* literals. The variables are checked in a prefixed ordering and those satisfying the two bounds are eliminated one by one. Different variable orderings can lead to eliminating different subsets of variables, as illustrated in Example 2.9.

Example 2.9 *Assume that* $\{x, y, A_1\}$, $\{\neg x, y, A_2\}$, $\{\neg y, A_3\}$, $\{\neg y, A_4\}$ *and* $\{\neg y, A_5\}$ *are all the clauses containing* x *and* y *in a CNF formula* ϕ*, where* A_1, A_2, A_3, A_4 *and* A_5 *are disjoint subsets of literals, and* clauseNbGrowthLimit = 1*. If we eliminate* x *first, all the clauses containing* y *in* ϕ *are* $\{y, A_1 \cup A_2\}$, $\{\neg y, A_3\}$, $\{\neg y, A_4\}$ *and* $\{\neg y, A_5\}$*. Then* y *can also be eliminated successfully without violating* clauseNbGrowthLimit.

However, if we eliminate y *first, all clauses containing* x *in* ϕ *are* $\{x, A_1 \cup A_3\}$, $\{x, A_1 \cup A_4\}$, $\{x, A_1 \cup A_5\}$, $\{\neg x, A_2 \cup A_3\}$, $\{\neg x, A_2 \cup A_4\}$ *and* $\{\neg x, A_2 \cup A_5\}$*. Then,* $|\phi_x \otimes_x \phi_{\neg x}| - |\phi_x| - |\phi_{\neg x}| >$ clauseNbGrowthLimit*. So,* x *cannot be eliminated after eliminating* y.

We review below some representative variable elimination procedures implemented in recent SAT solvers.

NiVER (Non-increasing Variable Elimination Resolution) [168] is a pre-processor for eliminating variables before solving ϕ by a CDCL solver. It implements the bounds *clauseNbGrowthLimit* and *resolventLengthLimit* using one constraint: the total number of literals in ϕ should not increase when eliminating a variable x. In other words, x is eliminated only if the total number of literals in $\{\phi \setminus \{\phi_x \cup \phi_{\neg x}\}\} \cup \{\phi_x \bigotimes_x \phi_{\neg x}\}$ does not exceeds the total number of literals in ϕ. Variables in ϕ are checked in their natural integer order and eliminated one by one if the above condition is met. This process iterates until no variable can be removed.

MiniSAT 2.2.0 [65] eliminates variables with $clauseNbGrowthLimit = 0$ in a pre-processing. The *incremental variable elimination* [146] improves the BVE of MiniSAT 2.2.0 by relaxing *clauseNbGrowthLimit* gradually in iterative applications of BVE. In the first round, $clauseNbGrowthLimit = 0$. Then, it takes value $8, 16, \ldots$ in the following rounds, respectively. In each round, the variables are checked and eliminated in increasing order of their $|\phi_x| \times |\phi_{\neg x}|$. A variable x is not eliminated if $|\phi_x \bigotimes_x \phi_{\neg x}| - |\phi_x| - |\phi_{\neg x}| > clauseNbGrowthLimit$ or any resolvent in $\phi_x \bigotimes_x \phi_{\neg x}$ contains more than *resolventLengthLimit*(=20 by default) literals.

Unlike NiVER and MiniSAT, Kissat [26] implements *in-processing* BVE to eliminate variables periodically (with increasing intervals) as the search proceeds. In each execution of BVE, variables are checked and eliminated in increasing order of $|\phi_x| \times |\phi_{\neg x}| + |\phi_x| + |\phi_{\neg x}|$ subject to the two bounds *clauseNbGrowthLimit* and *resolventLengthLimit*. As each in-processing BVE should be executed quickly, the *clauseNbGrowthLimit* bound is set to 0 and, then, if no variable can be eliminated in an execution, it is increased from 0 to 1, or multiplied by 2 in the next call to BVE, without exceeding 16. The initial *resolventLengthLimit* is set to 100 and increases gradually. In addition, a variable x is not eliminated if $|\phi_x| > 0$ and $|\phi_{\neg x}| > 0$, and $max(|\phi_x|, |\phi_{\neg x}|) > occlim$, where $occlim$ is initialized to 1000 and gradually increases. Finally, BVE stops if the total number of resolvents generated so far in this execution exceeds a limit.

The BVE procedure of MiniSAT and Kissat could be sketched in Algorithm 6, where ϕ contains the original clauses in the input formula (irredundant clauses in the terminology of Kissat), eventually vivified, after removing those clauses already satisfied, and L contains the clauses learnt so far in the CDCL search (redundant clauses in the terminology of Kissat), which is empty in a pre-processing. O is the increasing order of $|\phi_x| \times |\phi_{\neg x}|$ in MiniSAT and $|\phi_x| \times |\phi_{\neg x}| + |\phi_x| + |\phi_{\neg x}|$ in Kissat, respectively.

Note that eliminating a variable x can change $|\phi_y|$ and $|\phi_{\neg y}|$ of a variable y. Consequently, if y was checked before x, but was not eliminated because of the *clauseNbGrowthLimit* and *resolventLengthLimit* bounds, it is inserted into V to be checked again if $|\phi_y|$ or $|\phi_{\neg y}|$ is decreased by the elimination of x. Both in MiniSAT and Kissat, this insertion and sorting are achieved through a priority queue. V is organized using a priority queue, and sorted in $O(|V|log(|V|))$ time. The insertion of y into V is done in $O(log(|V|))$ time to keep V in order. Note that when x is eliminated, all learnt clauses containing x or $\neg x$ are lost.

Algorithm 6: BVE($\phi, L, O, growth, length$), a generic BVE procedure for CDCL solvers

Input: ϕ: a CNF formula not containing learnt clauses,
L: a set of learnt clauses,
O: a variable ordering,
$growth$: clauseNbGrowthLimit,
$length$: resolventLengthLimit
Output: $\phi \cup L$: a simplified CNF formula

```
1  begin
2      V ← list of variables in φ sorted in ordering O;
3      while V is not empty do
4          x ← the first variable in V;
5          Remove x from V;
6          if |φ_x ⊗_x φ_¬x| − |φ_x| − |φ_¬x| ≤ growth and no resolvent in φ_x ⊗_x φ_¬x
             contains more than length literals then
7              φ ← {φ \ {φ_x ∪ φ_¬x}} ∪ {φ_x ⊗_x φ_¬x};
8              for each variable y in φ_x ⊗_x φ_¬x do
9                  if y is not in V then
10                     insert y into V and sort V in O;
11             L ← L \ {c | c ∈ L, x ∈ c or ¬x ∈ c};
12             Remove subsumed clauses from φ ∪ L;
13     return φ ∪ L;
```

While in-processing BVE is very useful in solving SAT, it currently has two drawbacks: (1) when a variable is eliminated, all learnt clauses containing it are also removed, so that a part of the results in the CDCL search so far is lost; (2) an eliminated variable cannot be resolved on anymore in the subsequent CDCL search, hurting its performance. In Chapter **??** we propose a new variable ordering, called ESA (variable Elimination Scheduled by Activity), to remedy the above two drawbacks. In the ESA solvers, variables are sorted from small to large according to their heuristic. Consequently, the low activity variables satisfying the bounds are eliminated first in BVE, providing a soft protection to the high activity variables.

The ESA ordering is implemented in Cadical and three winning solvers based on Kissat. Since their execution condition for in-processing BVE are different and implement different decision heuristics, we can get a more complete picture of the effects of ESA. Experimental results demonstrate the effectiveness and robustness of ESA, and an empirical analysis is carried out to analyze the reasons. Besides, the principle behind ESA (i.e., high activity variables should be protected and reinforced) could be used to improve other solving techniques in SAT solvers, which we will also introduce in Chapter **??**.

2.4.2 Failed, Implied, Equivalent Literals Detection

Definition 2.12 (Failed Literal) *If propagating literal l derives a conflict, l is a failed literal and must be falsified.*

Definition 2.13 (Implied Literal) *If both propagating literal l and $\neg l$ satisfiy literal p, then p is an implied literal and can be satisfied under current assignment.*

Definition 2.14 (Equivalent Literal) *If l is satisfied, unit propagation satisfies p, and if l is falsified, unit propagation falsifies p, then l and p are equivalent literals and by replacing p with l we can remove p from the formula.*

Failed literal detection, implied literal detection and equivalent literal detection are three common simplification techniques in SAT solvers. The former two simplify the formula by assigning variables a value, and the last one achieve it by removing variables, thus all of them reduce the search space.

In practice, the three methods are combined, as all of them rely on unit propagation. Algorithm 7 show the detecting process. Before detection, the function Sort() arranges the variables in a specific order.

2.4.3 Clause Garbage Collection

Learnt clauses garbage collection is a crucial optimization technique used to manage memory, ensuring enough space for new clauses and improving solver performance. As modern CDCL SAT solvers often deal with large and complex problems, they generate a vast number of learnt clauses during the solving process. Over time, the accumulation of these clauses can consume significant memory resources and degrade the solver's efficiency. Clause garbage collection addresses this issue by strategically removing less useful clauses from memory.

Learnt clauses are added to prevent the solver from falling into the same conflicts. Some of them are critical for the solver's efficiency and cannot be deleted. There are several strategies to decide which clauses to remove:

- **Activity-Based Reduction:** In MiniSAT-based solvers, each learnt clause is assigned an activity score that reflects its importance. Clauses that participate frequently in recent conflicts (as conflict clauses or on the path to conflicts) are considered more active. Over time, the activity scores decay, ensuring that only recently useful clauses have high activity. In garbage collection, clauses with activity below a certain threshold are candidates for deletion.
- **Clause Length and LBD:** In Kissat-based solvers, the importance of clauses are measured by their length and LBD. Short clauses are generally more valuable because they contribute more directly to unit propagation. And clauses with lower LBD are in good quality because less decision variables are needed to deduce a conflict in them. The garbage collector preserves shorter clauses with lower LBD, while removing longer ones or those with higher LBD.

Algorithm 7: LiteralsDetection($V_{unassigned}$), a generic procedure for failed, implied, equivalent literals detection

Input: $V_{unassigned}$: the unassigned variables in the formula
Output: F: the set of failed literals,
I: the set of implied literals,
E: the set of equivalent literals

```
1  begin
2      Sort(V_unassigned);
3      F, I, E ← ∅ while V_unassigned ≠ ∅ do
4          l ← V_unassigned;
5          confl ← UnitPropagation(l);
6          if confl ≠ ∅ then
7              Backtracking();
8              F = F ∪ {l};
9          else
10             L_1 ← literals assigned during unit propagation;
11             Backtracking();
12             confl ← UnitPropagation(¬l);
13             if confl ≠ ∅ then
14                 Backtracking();
15                 F = F ∪ {¬l}
16             else
17                 L_2 ← literals assigned during unit propagation;
18                 Backtracking();
19                 I = I ∪ {l'}, l' ∈ L_1 ∪ L_2;
20                 E = E ∪ {l''}, l'' ∈ L_1 and ¬l'' ∈ L_2;
```

2.5 Conclusion

In this chapter, we focused on the SAT problem and its algorithms. We reviewed its applications and variants in the beginning. We also introduced the main complete and incomplete algorithms in SAT solving, with some formula simplification methods. In particular, we described the three main complete algorithms DP, DPLL and CDCL, along with the major techniques that are commonly used in conjunction with them, including unit propagation, branching heuristic among others. Then, in simplification methods we specifically presented bounded variable elimination, literals detection and learnt clauses elimination, as our work in Chapter 4 aims to improve these techniques. In the next chapter, we will provide a detailed review of Maximum Satisfiability (MaxSAT), the natural optimization extension of SAT.

3 Maximum Satisfiability

Sommaire

3.1 Definition and Variants . . . 52
3.2 Branch-and-Bound Algorithm . . . 53
3.2.1 Branch Heuristic . . . 55
3.2.2 Inference rule . . . 56
3.2.3 Lower Bound Estimation . . . 59
3.3 SAT-based Algorithms . . . 62
3.3.1 Model-improving . . . 62
3.3.2 Core-guided . . . 62
3.3.3 Implicit Hitting Set . . . 62
3.4 Incomplete Algorithm . . . 63
3.5 Conclusion . . . 67

In this chapter, we define the Maximum Satisfiability (MaxSAT) problem and its various variants in Section 3.1, establishing a foundational understanding of the topic. Following this, we present two primary complete methods for solving MaxSAT: the Branch and Bound (BnB) algorithm in Section 3.2, which systematically explores the solution space by constructing the binary assignment tree, and SAT-based approaches in Section 3.3, that leverage existing SAT solvers to handle MaxSAT instances effectively. Additionally, we provide a brief overview of incomplete algorithms that offer heuristic solutions, focus on their key techniques. This review sets the stage for understanding our contributions in the subsequent chapters.

3.1 Definition and Variants

The MaxSAT problem is an optimization version of the satisfiability problem (SAT), which is know to be NP-hard [149]. Given a MaxSAT formula ϕ in CNF format, generally represented as a set of clauses, solving ϕ involves finding a complete assignment (a solution) to all Boolean variables that maximizes (minimizes) the number of satisfied (falsified) clauses. Such assignment is called a optimal solution, and the other complete assignments are feasible solutions.

Example 3.1 *For the MaxSAT formula* $\phi = \{\{x_1\},\{x_2\},\{x_3\},\{\neg x_1,\neg x_2\},\{\neg x_2,\neg x_3\}\}$. *For the complete assignment* $\alpha_1 = \{\neg x_1,\neg x_2,\neg x_3)$, *the number of falsified clauses is 3; for the assignment* $\alpha_2 = \{x_1,x_2,x_3\}$, *the number of falsified clauses is; but for* $\alpha_3 = \{x_1,\neg x_2,x_3\}$,

the number of falsified clauses is 1. We can't find a better assignment, so α_3 is the optimal solution, and α_1, α_2 are feasible solutions.

MaxSAT has three variants: partial MaxSAT, in which some clauses, called hard clauses, cannot be falsified, and the others, called soft clauses, can be falsified; weighted MaxSAT, in which each clause has a positive finite weight representing the penalty of its violation; and weighted partial MaxSAT, that includes both hard clauses and weighted soft clauses. In weighted partial MaxSAT, a feasible solution must satisfy all the hard clauses, and an optimal solution is a feasible solution that minimizes the sum of weights of the falsified soft clauses.

Definition 3.1 (partial MaxSAT) *A partial MaxSAT formula contains two sets of clauses: the hard clauses set H, in which clauses must be satisfied, and the soft clauses set S, in which clauses can be falsified. Solving partial MaxSAT asks us to find a solution that satisfies all clauses in H and minimizes the falsified clauses in S. The plain MaxSAT problem is a partial MaxSAT problem with $H = \emptyset$.*

Definition 3.2 (weighted MaxSAT) *A weighted MaxSAT formual is a set of weighted clauses (cl, w), where cl is a clause and $w \in N^*$ is the associated weight. To solve weighted MaxSAT problem we should find an assignment that minimizes the sum of weights of falsified clauses. The plain MaxSAT problem is a weighted MaxSAT problem in which each clause has weight $w(cl) = 1$.*

Definition 3.3 (weighted partial MaxSAT) *The weighted partial MaxSAT is the combination of partial MaxSAT and weighted MaxSAT, its optimal solution satisfies all the hard clauses and minimizes the sum of weights of the falsified soft clauses.*

MaxSAT is important both theoretically and practically. Thanks to the efforts of the scientific community, MaxSAT has become a competitive generic approach to solve combinatorial optimization problems in real-world domains as diverse as scheduling [63, 29], maximum clique [110], FPGA routing [71], protein design [3], combinatorial testing [8], community detection in complex networks [86], diagnosis [60], group testing [53], planning [177], routing [126] and team formation [139], among others.

The significance of studying the Max-SAT problem, both in theory and practical solutions, stems from its strong connection to SAT and its wide range of applications. This importance is further emphasized by the annual Max-SAT evaluation events, held since 2006, which aim to assess the latest state-of-the-art Max-SAT solvers and encourage continuous advancements in their empirical performance.

3.2 Branch-and-Bound Algorithm

Early MaxSAT completion algorithms were mainly based on the branch-and-bound (BnB) method [31, 105]. The BnB algorithm in MaxSAT finds an optimal assignment by constructing a binary search tree in which each node corresponds to a partial

assignment. At each node, the current assignment is extended by inference rules. Then the algorithm calculates a lower bound (LB) on the number of falsified soft clauses, and compares it with the current upper bound (UB), which is the best solution found so far. If $LB \geq UB$, the current branch cannot yield a better solution, leading the solver to prune and backtrack; otherwise, the search continues. If a complete assignment is reached, we find a better solution, so that the UB and the optimal assignment are updated and a backtracking is performed. Otherwise, an unassigned variable is chosen by a specific branching heuristic. These steps are repeated until the whole search space is explored. Consequently, a tight LB help avoid useless search and thus is crucial for solving MaxSAT efficiently, BnB solvers rely on a variety of techniques and heuristics to get high quality LB. State-of-the-art BnB solvers include MiniMaxSat [78, 80], MaxSatz [106, 103, 102, 104], Ahmaxsat [1, 48], Akmaxsat [91], and (W)MaxCDCL [114, 115]. Algorithm 8 shows the outline of BnB algorithm for MaxSAT, in which:

- UB holds the current upper bound value, which gets updated each time a complete assignment is found. So that after fully exploring the search space, the value of UB will be the number of falsified clauses in the optimal solution. When initialization, UB can be determined by either counting the number of clauses in the input formula ϕ or calculating the cost of a random assignment. A more widely used approach is to generate the initial UB by a local search algorithm [31].
- α is the current assignment, it's a set of all literals assigned value $true$, and will be updated when branching, backtracking and executing inference rules.
- The function $Inference(\phi, \alpha)$ applies different inference rules on formula ϕ, the common inference rules are introduced in Section 3.2.2.
- The function $EstimateLB(\phi, \alpha)$ returns the estimation of lower bound under current assignment α. The simplest function returns the number of falsified clauses under α. Besides, many other estimation method was proposed, as we will explain in Section 3.2.3.
- The function $Learnt(\phi, \alpha)$ returns a clause as the reason of $LB \geq UB$, to prevent the solvers from falling into the same contradiction.
- $Backtrack(\phi, \alpha)$ cancels all the assignments made within and after the backtrack level.
- $BranchHeuristic(\phi)$ picks and assigns a literal, to branch on in the search tree.

Example 3.2 *For a MaxSAT formula* $\phi = \{(x_1), (x_2), (\neg x_2), (\neg x_1, x_3), (\neg x_1, x_2), (\neg x_2, \neg x_3)\}$, *the BnB algorithm on* ϕ *with the lexicographic branching ordering (i.e.,*$x_1 < x_2 < x_3$*) is shown in Figure 3.1[******]. The initial upper bound value is* $UB = |\phi| = 6$. *Assigning literal* x_1 *does not falsify any clause in* ϕ *and therefore we have* $LB = 0$. *Next, the*

Algorithm 8: BnB(ϕ), the Branch and Bound algorithm in MaxSAT

Input: ϕ: a CNF MaxSAT formula
Output: UB: the number of falsified clauses in the optimal solution

```
1  begin
2      UB ← |φ| α ← ∅ while true do
3          Inference(φ, α);
4          v ← EstimateLB(φ, α) if LB ≥ UB then
5              Learn(φ, α);
6              Backtrack(φ, α);
7          else
8              if |α| = |V| then
9                  UB ← LB;
10                 Backtrack(φ, α);
11             else
12                 l ← BranchHeuristic(φ);
13                 α ← α ∪ {l};
14         if α = ∅ then
15             break;
16     return UB
```

algorithms assigns the literals x_2 and x_3 leading to the falsification of the clauses $(\neg x_2)$ and $(\neg x_2, \neg x_3)$ and thus to the respective LB values 1 and 2. Since all the variables are assigned, the UB value is updated and becomes 2. When assigning $\neg x_3$ we have $LB = 2 \geq UB$ and thus a cut is performed. The algorithm continues to explore the search space while performing cuts when necessary until all possible assignments have been visited or discarded. Finally, the last UB value corresponds to the optimum of ϕ, i.e., $opt(\phi) = 2$, achieved by the assignment $\alpha = \{x_1, x_2, x_3\}$.

SAT inference rules which are sound for Max-SAT can also be used in BnB as will be showcased in the following section. **********

3.2.1 Branch Heuristic

Complete SAT solvers and branch-and-bound (BnB) solvers for MaxSAT share many similarities, especially in their use of branching heuristics. In fact, most branching heuristics used in (weighted) MaxSAT BnB solvers are either directly inspired by or adaptations of those developed for SAT solvers. Early MaxSAT solvers employed traditional heuristics like Jeroslow-Wang (JW) [87] and Maximum Occurrences in clauses of Minimum Size (MOMS)[1] [175, 70], often with slight modifications to adapt

[1] refer to Section **??**

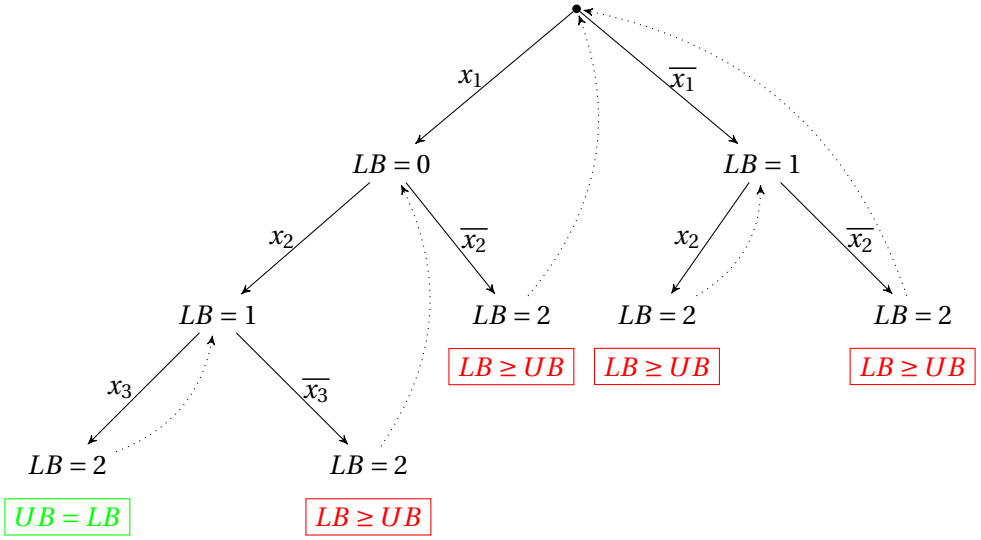


Figure 3.1: Search tree depicting the execution of a BnB algorithm for Max-SAT where branching is made in lexicographic order of the variables. Solid and dotted arrows represent respectively branching and backtracking.[*******]

them to the MaxSAT context [31, 4, 174]. These initial BnB solvers aimed to balance exploration with generating unit clauses to improve lower bound estimations, critical for branch-and-bound approaches.

More recent solvers have refined these traditional heuristics to better suit MaxSAT and weighted MaxSAT. The goal is to generate shorter clauses, particularly unit clauses, as they significantly improve lower-bound calculations. These solvers also handles opposed literal polarities, ensuring a more balanced search space. For instance, solvers like MaxSatz and Ahmaxsat build on MOMS but take into account the frequency of literals in unit, binary, and ternary [2] clauses. Additionally, Akamaxsat [107, 104] extends these heuristics by considering literals in inconsistent subsets of the formula during the lower bound estimation process, which will be explained in Section **??**. MiniMaxSat [78, 80], a notable weighted partial MaxSAT solver, uses a hybrid heuristic scheme in weighted partial Max-SAT. It applies VSIDS to hard clauses while employing a weighted version of the Jeroslow-Wang heuristic for soft clauses. Similarly, MaxCDCL [115, 114] adopts a hybrid approach, alternating between LRB and VSIDS phases, combined with different restart strategies (Luby restarts for LRB phases and glucose-style restarts for VSIDS phases).

3.2.2 Inference rule

The inference rules in MaxSAT are logical transformations or operations applied to simplify the formula without changing the optimal solution. These rules are used in each node of the BnB search tree, and help in reducing the problem size or in deriving

[2] clauses that contain exactly three literals

new information that speeds up the solving process. They are also used to estimate a better lower bound. Semantic rules depend on the truth assignments of the formula, ensuring further assignments. However, syntactic rules focus on the manipulation of the formula's structure without considering specific assignments, they are applied to simplify the formula while preserving the equivalence of the problem, which can make MaxSAT solving more efficient. Besides them, there are also preprocessing methods and clause learning techniques.

It is important to note that even though they are similar, the inference rules for SAT usually cannot be used in MaxSAT, as they preserve satisfiability but cannot preserve the minimum number of falsified clauses. For example, unit propagation, which is a successful SAT solving techniques, is unsound in MaxSAT, as illustrated in Example 6.1. The same applies to SAT resolution.

Example 3.3 *For a MaxSAT formula* $\phi = \{(x_1), (x_2), (x_3), (\neg x_1, \neg x_2), (\neg x_1, \neg x_3)\}$. *The optimal solution of* ϕ *is* $(0,1,1)$ *with only one falsified clause* (x_1). *But by propagating* x_1, *both* $\neg x_2$ *and* $\neg x_3$ *are propagated and there are two falsified clauses in the formula. The result shows that unit propagation is unsound in MaxSAT.*

In the following we will introduce the most common semantic rules for MaxSAT first. Then, the syntactic rules represented by the MaxSAT resolution rule. Finally, we will briefly mention other inference rules.

Pure Literals A pure literal is a variable that appears with only one polarity, it can be satisfied without making any clause false. This rule was first used in SAT solving [****], and it's also valid in MaxSAT. Applying this rule surely extend the current assignment.

Hard Unit Propagation [***] This rule is applied when $LB = UB - 1$, where all unit soft clauses $cl = (l)$ should be satisfied since falsifying them won't lead to a better assignment. This process is also called hardening. In the weighted MaxSAT, the hardening may happen at any node when there is a soft clause with $w(cl) \geq UB - LB$.

Dominant Unit Clause [***] This rule consists in satisfying literal l if the number of unit soft clauses $cl = (l)$ is not less than the number of clauses $cl = (\neg l)$. In the weighted case, the measurement is the sum of weights of this kind of clauses.

The syntactic inference rules mainly include the MaxSAT resolution rule [***], which is the extension of the SAT resolution rule, and its derived rules. Same as the unit propagation for SAT, the resolution rule for SAT is also unsound in MaxSAT. The difference between them lies in their objectives: SAT aims to determine whether a formula is satisfiable, whereas MaxSAT aims to maximize the number of satisfied clauses. Some relevant studies analyze how a refinement of SAT resolution rule, can be applied in MaxSAT solving [30, 77, 95, 107, 2]. The extended resolution rule produces a resolvent clause and a set of compensation clauses, which are used to preserve equivalence, i.e, keeping the number of maximal satisfied clauses.

MaxSAT Resolution Rule Given two clauses $cl_1 = (x, a_1, ..., a_s)$ and $cl_2 = (\neg x, b_1, ..., b_t)$, by MaxSAT Resolution Rule we get their resolvent $(a_1, ..., a_s, b_1, ..., b_t)$ and $s + t$ compensation clauses, as shown below. By adding these new produced clauses, we remove clauses cl_1 and cl_2. The rule is proved complete: given a MaxSAT formula ϕ whose minimum number of falsified clauses is m, with the application of MaxSAT resolution we get a new formula ϕ' whose minimum number of falsified clauses is exactly m.

$$\frac{cl_1 = (x, a_1, ..., a_s) \quad cl_2 = (\neg x, b_1, ..., b_t)}{\begin{array}{c} cl_r = (a_1, ..., a_s, b_1, ..., b_t) \\ ccl_1 = (x, a_1, ..., a_s, \neg b_1) \\ ccl_2 = (x, a_1, ..., a_s, b_1, \neg b_2) \\ \vdots \\ ccl_t = (x, a_1, ..., a_s, b_1, \ldots, b_{t-1}, \neg b_t) \\ ccl_{t+1} = (\neg x, b_1, ..., b_s, \neg a_1) \\ ccl_{t+2} = (\neg x, b_1, ..., b_s, a_1, \neg a_2) \\ \vdots \\ ccl_{t+s} = (\neg x, b_1, ..., b_s, a_1, \ldots, a_{s-1}, \neg a_s) \end{array}}$$

where cl_r is the resolvent clause and $ccl_1, ..., ccl_{t+s}$ are compensation clauses.

Neighborhood Resolution [<empty citation>] It's a specific case of MaxSAT resolution, which is defined below, where the two resolution candidate clauses have the same literals except x and $\neg x$.

$$\frac{cl_1 = (x \vee A) \quad cl_2 = (\neg x \vee A)}{A}$$

where A is the disjunction of literals.

Chain Resolution [<empty citation>] It's used in deducing empty clause without increasing the formula size. This rule is mainly used in lower bound estimation, as we will illustrate in Section **??**, besides, it's also applied in core-guided MaxSAT solving. For $k \geq 1$, we have:

$$\frac{(l_1), (\neg l_1, l_2), (\neg l_2, l_3), \ldots, (\neg l_{k-1}, l_k), (\neg l_k)}{\square, (l_1, \neg l_2), (l_2, \neg l_3), \ldots, (l_{k-1}, \neg l_k)}$$

Cycle Resolution [<empty citation>] This rule aims to resolve specific types of relationships between clauses where there is a cycle formed by soft clauses in a certain structure, allowing for simplifications. For $k \geq 2$, we have:

$$\frac{(\neg l_1, l_2), (\neg l_2, l_3), \ldots, (\neg l_{k_1}, l_k), (\neg l_1, \neg l_k)}{(l_1), (l_1, \neg l_i, l_{i+1})_{2 \leq i < k}, (\neg l_1, l_i, \neg l_{i+1})_{2 \leq i < k}}$$

The above resolution rules all apply to unweighted MaxSAT, but they can also be extended to weighted MaxSAT, as introduced in [****]. In fact, in the case where clauses have associated weights, we have the fold and unfold rules that help to reduce the complexity of the problem. Folding involves combining soft clauses that have the same literal or a combination of literals but different weights. The goal is to reduce the number of clauses by folding them into a single clause with a cumulative weight. Unfolding is essentially the reverse of folding. It's used when a clause has a large weight and can be broken down into multiple clauses with smaller weights, each representing a part of the original clause.

Fold & UnFold Rule For two clauses cl with weight w_1 and cl with w_2, the fold rule is shown in the left. For a single clause cl with weight $w_1 + w_2$, the unfold rule is in the right.

$$\frac{(cl, w_1) \quad (cl, w_2)}{(cl, w_1 + w_2)} \qquad \frac{(cl, w_1 + w_2)}{(cl, w_1) \quad (cl, w_2)}$$

In addition to those inference rules mentioned above, we also have mechanisms for preprocessing, which can be used before MaxSAT solving to simplify the formula, and techniques for learning clauses from conflicts. Since the research on the latter is in parallel with the development of lower bound estimation, we will introduce it in the following section.

3.2.3 Lower Bound Estimation

In MaxSAT and particularly Weighted MaxSAT, lower bound estimation is crucial for optimizing the solving process. The lower bound provides a minimum cost (or number of falsified clauses) that must be incurred given the current assignment of the problem. This helps the solver determine how far it is from an optimal solution and whether certain branches of the search space can be pruned. The earliest estimation of LB is the number of falsified clauses in the unweighted formula ϕ under current assignment α, or the sum of weights of falsified clauses in the weighted formula ϕ. Since unweighted clauses can be seen as clauses with weights 1, without loss of generality, we say $LB = falsifiedWeight$ in such case. This simple approach only considers clauses that are already falsified, and usually does not work well. The researches after refine it by computing the number of *cores*, also referred as disjoint inconsistent subsets, in the current formula $\phi|_\alpha$.

Definition 3.4 (Core) *A core is a subset of soft clauses that cannot all be satisfied without falsifying a hard clause at the current search tree node. It is represented by core* $c = \{cl_1, cl_2, \ldots, cl_n\}$*, where* $cl_1, cl_2, \ldots, cl_n$ *are the set of clauses that cannot all be satisfied. If a core is found within a specific branch of the search tree, i.e., it relies on a specific assignment, we call it local core, otherwise it is global core.*

Different methods for detecting disjoint cores have been described in the literature. The first method, proposed by Wallace and Freuder [**wallace1993comparative**], involves counting the number of pairs of complementary unit soft clauses, (x_i) and $(\neg x_i)$, where each pair constitutes a core. Shen and Zhang [**shen2004study**] improved this approach by using linear resolution to generate additional unit clauses before counting. Alsinet et al. [**alsinet2004max**] introduced the use of the star rule to detect sets of clauses of the form $\{(x_1), ..., (x_k), (\neg x_1, ..., \neg x_k)\}$, with each set representing a core.

MaxSatz [**li2005exploiting**, **li2006detecting**] was the first BnB MaxSAT solver that uses unit propagation (UP) to detect cores. It repeatedly propagates a unit clause l, i.e., satisfying l (and falsifying $\neg l$), removing all clauses containing l and deleting $\neg l$ from the remaining clauses until there is no unit clause or an empty clause cl is produced (all literals of cl are falsified and deleted). When an empty clause cl is produced, an implication graph can be constructed from cl. The detected core is the set of the soft clauses in a path to cl in the implication graph. Figure 3.2 shows an example of UP-based core detection.

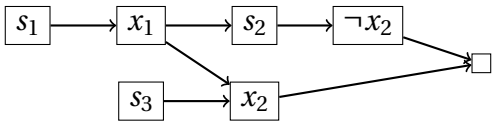


Figure 3.2: An example of implication graph to identify a core. The formula ϕ contains 3 soft clauses: (s_1), (s_2), (s_3), and 4 hard clauses: $(\neg s_1, x_1), (\neg x_1, s_2), (\neg s_2, \neg x_2)$ and $(\neg s_3, \neg x_1, x_2)$. Each node represents a satisfied literal l. With the incoming edges, it represents also the hard clause allowing to propagate l. For example, the node x_2 represents the clause $(\neg s_3, \neg x_1, x_2)$. Propagating (s_1) satisfies x_1, s_2 and $\neg x_2$, but propagating (s_3) satisfies x_2, making $(\neg s_2, \neg x_2)$ or $(\neg s_3, \neg x_1, x_2)$ empty. By retracing the implication graph from the empty clause to the propagated soft clauses, we know that (s_1) and (s_3) cannot be satisfied at the same time. So, the core $\{(s_1), (s_3)\}$ is identified. Note that s_2 is propagated because of the hard clause $(\neg x_1, s_2)$ and does not belong to the identified core.

The core detection in MaxSatz depends on the existence of soft unit clauses in ϕ, whose number is limited. MaxCDCL introduces a soft literal s for each non-unit soft clause cl by adding hard clauses encoding $s \leftrightarrow cl$ to ϕ, so that all soft clauses become unit and can be propagated.

Definition 3.5 (Soft Literal) *A soft literal is a literal s that represents a soft clause cl, i.e., s is satisfied if and only if the corresponding soft clause cl is satisfied. To ensure the equivalence of soft literal and soft clauses, when introducing s we should also add a group of hard clauses into the formula. For example, for a soft clause $cl = (x_1, x_2, \ldots, x_n)$, when replacing cl with a soft literal s, we should add $n+1$ clauses $\{(\neg s, x_1, x_2, \ldots, x_n), (s, \neg x_1), \ldots, (s, \neg x_n)\}$ into the formula.*

In unweighted MaxSAT, there is at least one falsified clause in one core if extending the current assignment to a complete assignment. The existing unweighted MaxSAT BnB solvers lock the soft clauses in the already detected cores so that they cannot be used to detect additional cores, ensuring the disjointness of the cores. So that the number of cores implies the minimal number of falsified clauses in current branch, which we call $estimatedWeight$, as the falsified number can be seen as the falsified weight in the unweighted case.

****

In weighted MaxSAT, these solvers lock a part of the weight of the soft clauses instead of the soft clauses themselves. For example, let $c = \{cl_1, cl_2, cl_3\}$ be a detected core and $w(cl_1) = 2$, $w(cl_2) = 3$, $w(cl_3) = 6$. So, $w(c) = 2$. The weight of cl_2 (cl_3) is split to lock its weight 2, but its remaining weight 1 (4) can be used to detect additional cores. The detected cores are considered to be disjoint in a larger sense, because cl_2 (cl_3) can be considered as two distinct copies, one with weight 2 that is locked in c to decide $w(c)$, the other with weight 1 (4), referred as remaining weight and denoted by $rw(cl_2)$ ($rw(cl_3)$) in the general case, that can be used to detect new cores.

We use $wlock(cl, c)$ to denote the weight of cl locked into core c, which is the minimum weight locked into c to ensure $w(c)$ and cannot be used to detect other cores in existing BnB solvers. In the above example, $wlock(cl_1, c) = wlock(cl_2, c) = wlock(cl_3, c) = 2$ ensures $w(c)$=2. If $wlock(cl, c) < 2$ for any $cl \in \{cl_1, cl_2, cl_3\}$, then $w(c) < 2$.

The function to underestimate $estimatedWeight$ is commonly referred to as *lookahead*. It computes $estimatedWeight$ by detecting disjoint local cores using unit propagation.

To speed up the search, the solver computes LB = $falsifiedWeight + estimatedWeight$ before branching, where $falsifiedWeight$ is the sum of weights of the soft clauses falsified under current partial assignment α, and $estimatedWeight$ is an underestimation of the sum of weights of the soft clauses that will be falsified if α is extended to a complete assignment. Let UB denote the total weight of falsified soft clauses in the best solution found so far. If LB≥UB, no better solution can be found in the subtree rooted at the current node, which is then pruned.

Therefore, a crucial factor in the performance of a BnB solver is the accurate computation of $estimatedWeight$, which is typically achieved by detecting disjoint inconsistent subsets of soft clauses under α, called local *cores* w.r.t. α. As this manuscript exclusively considers local cores, we will omit the word "local" in the sequel. Each core c is associated with a weight $w(c)$, defined as the minimum total weight of its soft clauses that are falsified across all possible extensions of α. In existing BnB MaxSAT solvers, the core weight is always 1 for unweighted MaxSAT, while for weighted MaxSAT, it is the minimum weight among its soft clauses. Since the cores are disjoint, $estimatedWeight$ is the sum of the weights of these cores.

Definition 3.6 (Local Core) *A local core is a subset of soft clauses that cannot all be satisfied without falsifying a hard clause at the current search tree node.*

3.3 SAT-based Algorithms

SAT-based algorithms convert the MaxSAT optimization problem into a series of SAT decision problems and then iteratively call the SAT solver to solve them. This type of complete algorithm can also be easily converted into an incomplete version by outputting the better feasible solution in real time [16, 148]. SAT-based algorithms can be mainly divided into three types: Model-improving, Core-Guided, and Implicit Hitting Set.

The SAT-based approaches transform MaxSAT problems to a series of decision problems and solve them by SAT solvers. These algorithms has become dominant over the past few decades because they can make full use of the clause learning in SAT solvers, thus show high efficiency in industrial instances. There are three different SAT-based algorithms we will present in the following: the Model-improving algorithm,

3.3.1 Model-improving

Solving a MaxSAT problem ϕ aims to find a minimal number of falsified clauses in ϕ. It can be transform to a series of problems that given a $cost$ and determine if the falsified clauses number in ϕ is less than the $cost$. If the answer is yes, then $cost = cost - 1$ and comtinues.

The MaxSAT solvers equipped with Model-improving approach include Sat4j [96], QMaxSAT [90] and Pacose [150]. Recently, it is also used in imcomplete MaxSAT solvers as each completed call to the SAT solver returns an improved solution.

3.3.2 Core-guided

The core-guided algorithm is an approach that iteratively modifies the lower bound of the solution [5]. Initially, the algorithm aims for the SAT solver to satisfy the entire formula, including both hard and soft clauses. If the formula is unsatisfiable, the SAT solver is asked to output an unsatisfiable core composed of soft clauses. The algorithm then relaxes this unsatisfiable core by introducing relaxation variables and constraints. After that, the modified formula is fed back to the SAT solver, and the process of finding and relaxing unsatisfiable cores is repeated until the SAT solver returns a "satisfiable" result.

The first SAT-based algorithm is the core-guided algorithm, which provided a new approach to solving partial MaxSAT problem [72]. Subsequently, core-guided algorithms that can be used to solve weighted partial MaxSAT were proposed [6, 137], so did the various methods for simplifying constraints [7].

3.3.3 Implicit Hitting Set

The implicit hitting set algorithm also utilizes a SAT solver to find unsatisfiable cores [156]. Unlike the core-guided algorithm, which returns only one unsatisfiable core at a time, the implicit hitting set algorithm immediately relaxes the unsatisfiable core

after finding it and continues solving with the SAT solver. This process is repeated until the SAT solver returns a "satisfiable" result.

During this process, the SAT solver extracts lots of unsatisfiable cores. The algorithm then uses an integer programming (IP) solver to find the minimum hitting set for this set of unsatisfiable cores. A hitting set is a subset of soft clauses such that at least one soft clause from each unsatisfiable core is included in this set. The goal is to find the hitting set where the sum of the original weights of the selected soft clauses is minimized.

In the initial implicit hitting set algorithm, unlike model-improving and core-guided algorithms, cardinality constraints were not used. As a result, the algorithm performed poorly on certain instances where variables exhibited symmetry. To address this issue, Berg et al. [15] proposed introducing count variables, which significantly improved the performance of the implicit hitting set algorithm in solving such cases.

3.4 Incomplete Algorithm

Local search (LS) algorithms are the most representative incomplete algorithms for solving MaxSAT. They work well when solving random instances. However, for a long time, LS algorithms could not solve industrial instances as effective as SAT-based algorithms. This phenomenon has been changed recently with the introduction of some advanced local search algorithms [51, 97, 52]. Moreover, LS methods were also used as optimization components by the complete MaxSAT solvers [55, 56].

The MaxSAT local search algorithm explores the solution space by searching the neighborhood, to find high-quality solutions. Most existing MaxSAT local search algorithms follow the process shown in Algorithm 9. The algorithm first initializes an assignments α of the input MaxSAT formula ϕ. The simple initialization method is random initialization, that is, each variable is randomly assigned 1 or 0, and then α is used to initialize the optimal solution α^* (line ***). Subsequently, the algorithm repeats the operations of selecting variables and flipping (lines ***) until the stop condition is reached (such as reaching the time limit). During this process, if a better solution is found, α^* is updated (line ***). Finally, the algorithm returns α^* if it is a solution, otherwise it returns UNKNOWN.

Based on this framework, many studies have proposed a large number of improvement methods for different components, mainly including initialization methods, variable selection strategies, and clause weighting techniques. We will introduce them in order.

Initialization Function Initialization() is used to generate an initial solution. Since the size of instances are large (more than 10 million variables or clauses) , but the solving time are limited in LS solving, the MaxSAT local search algorithm can only explore a rather limited solution space. Therefore, the initial solution determines the area in which the local search algorithm explores the solution space, significantly affecting the performance of the algorithm.

Algorithm 9: LocalSearch(ϕ), the local search algorithm in MaxSAT

Input: ϕ: the maxsat formula
Output: α^*: the best solution found; otherwise "UNKNOWN"

```
1  begin
2      α ← Initialization();
3      α* ← α; \\the best solution so far
4      while the ends condition does not meet do
5          v ← VariableHeuristic();
6          Flip the assignment of v in α;
7          if cost(α) < cost(α*) then
8              α* ← α;
9      if α* is a solution then
10         return α*;
11     else
12         return UNKNOWN;
```

The simplest initialization method is random initialization, that is, randomly assigning a true or false value to each variable. This method is widely used in various early MaxSAT local search algorithms [161, 171], and even some recent work [133, 39] also use such method.

Based on random initialization, some research introduced the idea of Backbone to assist in generating better initial solutions [170, 179]. In MaxSAT, the backbone of a formula can be regarded as a set of variables whose values are consistent in all optimal solutions (there may be multiple optimal solutions). In practice, it is very difficult to extract the backbone. A common method is replacing the set of optimal solutions by the set of local optimal solutions, and extracting information from the latter. Those information are used in adjusting the probability of random initialization of variables.

Recently, a decimation initialization method based on unit propagation was proposed [40]. At each step, the method checks whether there is a unit hard clause or a unit soft clause in the current CNF formula. If so, the unit propagation is performed prioritily on hard clauses and the formula is simplified. Otherwise, an unassigned variable is randomly selected and assigned, and the formula is simplified. The above operation is repeated until all variables are assigned. This method satisfies clauses with shorter lengths first, i.e., clauses with stronger constraints, so that guarantees the quality of initial solution.Meanwhile, it also takes into account some randomness, so that the algorithm can search in different areas of the solution space each time it restarts [52, 161]. This method improves the performance of LS algorithms Dist [39] and CCEHC [133]. And it has also been used by various subsequent algorithms [97, 183, 52].

In addition, some LS algorithms [152] use SAT solvers to generate initial solutions. They input the set of hard clauses into the SAT solver to obtain a partial assignment

that satisfies all hard clauses, and then assign the remaining variables. The generated initial solution is definetely a feasible solution of MaxSAT, but LS algorithms equipped with this method lose are not simple and portable anymore.

Variable Heuristic This function selects variables and flips them at each search step. It is crucial to the efficiency of MaxSAT LS algorithm. Many research are focus on selecting the variables to flip.

The performance of variable selection strategy is closely related to the metric for evaluating the benefits of flipping variables. The most commonly used evaluation metric is the score function, and it has been used in local search algorithms of SAT and MaxSAT for a long time until today [161, 163, 45]. The score function $score(v)$ in the MaxSAT local search algorithm evaluates the immediate benefit of flipping the variable v. Normally, it can be defined as the amount by which the sum of dynamic weights of unsatisfied clauses decreases after flipping v, or the amount by which the sum of dynamic weights of satisfied clauses increases. The higher the variable score, the higher the priority of flipping it. When there is no variable with a score greater than 0, the search is considered to be trapped in a local optima.

The local search algorithm usually greedily flips the variable with the highest score, but a purely greedy strategy makes the algorithm falling into the local optima and cyclic search. Therefore, many LS algorithms use the random walk strategy to escape from the local optima [163, 45]. For example, an unsatisified clause is randomly selected with probability p, then, with probability $1-p$, one of its variables is randomly selected and flipped based on their score to satisfy the clause. This strategy has also been widely used in a series of recent advanced MaxSAT local search algorithms [97, 183, 52].

Another widely used strategy is a probability sampling-based technique called Best from Multiple Sampling (BMS) [34]. When the MaxSAT problem is large, the number of candidate variables at each step may be huge, and it is time-consuming to select the variable with the highest score from them. The BMS technique first samples k variables from the candidate variables, and then selects the variable with the highest score among them as the flipping variable. In fact, BMS is a technique that balances the accuracy and the efficiency in search. It has certain statistical theoretical guarantees. Assuming that the number of samples $k = 50$, the probability that the score of the final selected variable is in the top 10% of all candidate variables is $1 - 0.9^{50} > 0.9948$. This probability is only related to k but not n, which means that no matter how large n is, the time complexity can be reduced from $O(n)$ to $O(1)$ by sacrificing a small amount of accuracy. The BMS technique was first proposed in a LS algorithm for solving the minimum vertex cover problem [34]. Now this technique has been widely used to solve various combinatorial optimization problems including MaxSAT [97, 173, 46].

In addition, configuration checking is a technique similar to the taboo search [75]. It was first used to solve the minimum vertex cover problem [44] and the SAT problem [44]. Then, it was used in some local search algorithms of MaxSAT [133, 134]. The core idea is that, after flipping variable x, if the value of all its neighbor variables (the other variables appearing in the same clause as x) have not changed, then the variable

cannot be flipped, to avoid the cyclic search.

Clauses Weighting Clause weighting is a core technology of MaxSAT local search algorithm. It assigns dynamic weights to clauses and adjusts them during the search. The idea is to give higher dynamic weights to clauses that are more difficult to satisfy, and lower dynamic weights to clauses that are easier to satisfy, so that the algorithm prefers satisfying difficult clauses. In clause weighting, the variables scores are related to the dynamic weights of clauses, so that a reasonable adjustment of the weight guides the score of the variables, thereby changing the search direction and helping the algorithm escape from the local optima, and find a better solution.

Clause weighting technology has been used to solve SAT and MaxSAT [45, 69] for a long time. Its basic idea is that when the LS is trapped in the local optima, i.e., when there is no variable with a score greater than 0, the weighs of unsatisfied clauses under current solution is increased. In recent years, as more general problems such as partial MaxSAT and weighted partial MaxSAT have been favored by academia and industry, various complex clause weighting design for the different properties of hard and soft clauses have emerged. For example, in the Dist algorithm [39], only hard clauses have dynamic weights, while soft clauses have the original weights from instances. When calculating the variable score, the hard score and soft score are considered separately, and the priority of hard score is higher than the soft score when selecting the flipping variables. The same method used in the CCEHC algorithm [133].

Subsequently, the SATLike algorithm [97] weights both hard and soft clauses and uniformly calculate the score of variables, thus simplifying the evaluation of variables. It sets the initial dynamic weight of hard clauses to 1 and the initial dynamic weight of soft clauses to their original weight. In weighted partial MaxSAT problems, the initial priority of soft clauses with weight greater than 1 is higher than that of hard clauses, which makes it easier to solve those instances where hard clauses are easier to satisfy but soft clauses are harder to satisfy. SATLike further sets the increment of the dynamic weight of hard clauses to be approximately 3 times that of soft clauses, so that as the search continues, the priority of hard clauses will gradually exceed that of soft clauses, ensuring the algorithm to find a feasible solution. The performance of SATLike far superior to Dist and CCEHC. And the improved version, SATLike3.0 [35], further proposes that when the algorithm fails to find a feasible solution, the dynamic weight of the soft clause is initialized to 0. This initialization method is also used by subsequent local search algorithms [183, 52].

Recently, the NuWLS algorithm [52] has further solved some defects in the SATLike algorithm, that is, for different instances, the initial dynamic weight of the soft clause is extremely unstable, causing the algorithm performance to be unstable, too. NuWLS normalizes the original weights of soft clauses, so that the algorithm has relatively stable performance for various MaxSAT instances. In the NuWLS-c-2023 [51], the improved NuWLS local search algorithm proposes to uniformly weight soft clauses and keep the ratio between the dynamic weights of soft clauses to be the same as the original weights. This method further optimizes the overall performance of the local search algorithm.

Currently, the clause weighting technology in the MaxSAT local search algorithm is well developped. The algorithm can even avoid cyclic search and the local optima by only using clause weighting, without any taboo search or configuration checking technology. Improvements in clause weighting technology have also created a series of champion solvers in the MSE anytime track from 2018 to now [98, 50, 51].

3.5 Conclusion

In this chapter, we defined the MaxSAT problem and introduced its variants and applications. We also reviewed the major algorithms for MaxSAT solving, including the Branch and Bound methods (BnB) and SAT-based approaches which are complete methods, and the local search methods which are incomplete. In particular, we provided a detailed review on the lower bound estimation methods in BnB solvers, as our contributions in Chapter 5 aims to provide further understanding of it and improve the quality of computered lower bound. This chapter concludes the background section of this manuscript, our contributions will be presented in the following 3 chapters.

Contributions

4 A New Variable Ordering for In-processing Bounded Variable Elimination in SAT Solvers

Sommaire

4.1 Introduction & Motivation 69
4.2 ESA: Variable Elimination Scheduled by Activity 71
4.2.1 The Principle of ESA 71
4.2.2 Implementation of ESA in Cadical and Kissat Based Solvers . . . 73
4.3 Experimental Results 74
4.3.1 Effectiveness of the ESA Procedure 74
4.3.2 Analyzing the Behaviour of ESA 77
4.4 Other Implementations 79
4.4.1 Keep Binary Learnt Clauses after Variable elimination 79
4.4.2 Variable Heuristic in Other Simplification Functions 79
4.5 Solvers at SAT Competitions 81
4.6 Conclusion 82

This chapter includes our work in SAT solving, that is, using variable heuristic score to guide simplification methods. The improved methods include bounded variable elimination, failed/implied/equivalent literals detection, and learnt clauses elimination. In Section 4.1 we introduce the motivation of applying variable heuristic score in simplifications, especially in bounded variable elimination. The implementation on bound variable eliminations and results can be found in Section 4.2 and 4.3. In Section 4.4 we introduce the application of variable heuristic score in other simplification methods. We submitted all these solvers to the SAT competition, the ranking of them are listed in Section 4.5. Finally, we summary our work in 4.6.

4.1 Introduction & Motivation

The state-of-the-art SAT solvers used to simplify the SAT formula by variable elimination in pre-processing. But Kissat [26], which is the best solver in the SAT competition 2020 and the baseline for other solvers since then, uses variable elimination as a in-processing function. The solver Kissat starts at CDCL search, when certain conditions are met, it then turns to execute variable elimination or other simplification methods.

As shown in Algorithm 10. The entry conditions of these simplification methods are carefully designed to prevent them from being executed too frequently, as each execution would incur additional overhead.

Algorithm 10: Kissat(ϕ), the simplified solving process of the kissat solver

```
Input: φ: a CNF SAT formula
Output: SAT or UNSAT: the satisfiability of φ
1 begin
2     while true do
3         CDCL(φ);
4         if meets the condition of elimination then
5             VariableElimination(φ);
6         if meets the condition of other simplifications then
7             Simplification(φ);
```

The bounded variable elimination is based on regular resolution, but the CDCL search, which from Beame's research, is equivalent to unrestricted resolution under a few assumptions [13]. So that the alternation of BVE and CDCL can be seen as an alternating between regular resolution and unrestricted resolution, which have different capacity to solve SAT formulas. To be more clear, unrestricted resolution is exponentially stronger than regular resolution by allowing the variables to be resolved on many times, as illustrated in Example 4.1 and also proved in theory [84, 76].

Example 4.1 *Given a unsatisfiable SAT formula ϕ with ten clauses (x, x_1, x_2), $(\neg x, x_1, x_2)$, $(\neg x_1, x3)$, $(\neg x_2, x_4)$, $(\neg x_5, x)$, $(\neg x_6, \neg x)$, $(\neg x_3, x_5, x)$, $(\neg x_4, x_5, x)$, $(\neg x_3, x_6, \neg x)$, $(\neg x_4, x_6, \neg x)$. It takes eleven steps for regular resolution to prove its unsatisfiability (as shown in Figure 4.1), but only ten steps are needed for unrestricted resolution (Figure 4.2).*

The process of CDCL aims to find conflicts as soon as possible, so it relies on variables with high heuristic scores (which means they occur frequently in recent conflicts). But bounded variable elimination follows a wholly different rule, that it, eliminating variables that brings minimal clauses increase first. When BVE selects variables for elimination, it does not consider the variable's heuristic score, so highly active variables may also be deleted, which will affect the execution effect of the next CDCL search. Besides, those high scores variables also have a higher probability to occur in good learnt clauses. Eliminating them also deletes those learnt clauses they related to, so that hurts CDCL.

The two limitations above inspire us to use a different measurement to guide variable elimination. The new method needs to maintain the performance of BVE, but try not to hurt the CDCL search. We will introduce it in the next section.

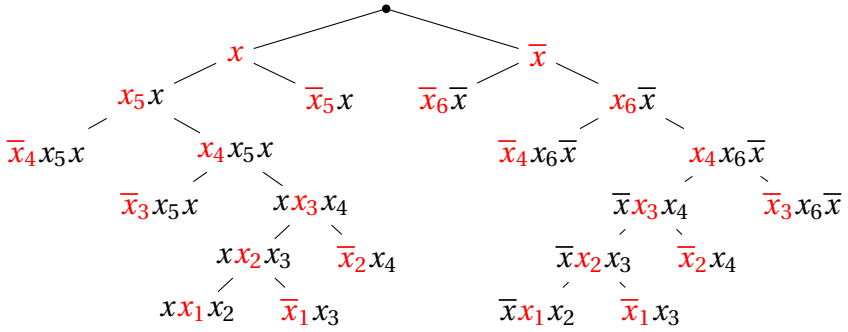


Figure 4.1: The resolution tree of regular resolution. From the leaf nodes to the root node, the algorithm performs regular resolution on the variables $x_1,\ldots,x_6,x$ in order. The pivot variable at each resolution are marked red. As can be seen, no variables are reused, and it takes 11 steps for the algorithm to prove the unsatisfiability.

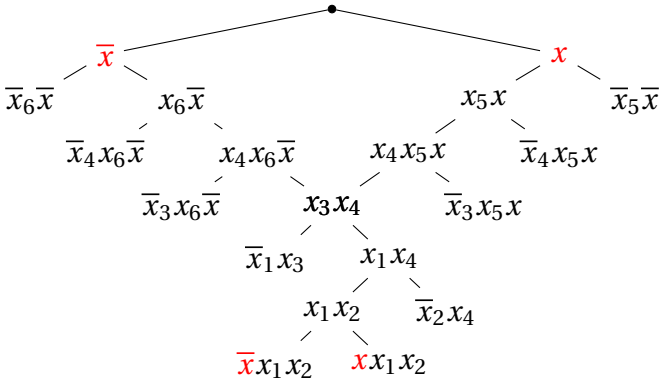


Figure 4.2: The resolution tree of unrestricted resolution. The variable x is used twice on the path to the root node, it is marked red. Only 10 steps are needed for the algorithm to prove the unsatisfiability.

4.2 ESA: Variable Elimination Scheduled by Activity

It is well-known that the variables in a CNF formula are not equally important for solving it. In this section, we define a new variable ordering called *ESA* (variable Elimination Scheduled by Activity) to eliminate first the variables of smaller importance, which can reinforce and prevent the variables of greater importance from being eliminated by increasing the number of clauses containing them. We first describe the principle of ESA, and then its implementation in Cadical and three Kissat based solvers.

4.2.1 The Principle of ESA

Since CDCL under a few assumptions is shown to be equivalent to general (or unrestricted) resolution and BVE belongs to a process based on regular resolution, a

CDCL solver implementing BVE could be thought as hybridizing or switching between unrestricted resolution and regular resolution. Recall that unrestricted resolution is exponentially stronger than regular resolution by allowing to resolve on the same variables several times in any path of a resolution tree. So, in theory, BVE could hurt the power of CDCL, because the learnt clauses containing the eliminated variables are lost and the eliminated variables cannot be resolved on anymore.

In practice, however, not all variables need to be resolved on several times to produce a short resolution proof when solving a large CNF formula. Indeed, we observe that a state-of-the-art CDCL solver could never need to resolve on a subset of variables when solving some CNF formulas: these variables never participate in any conflict nor in any learnt clause.

But what are the variables a state-of-the-art CDCL solver will resolve on several times? The answer is probably those variables participating frequently in recent conflicts, i.e., the variables with high activities. There are two reasons: (1) the solver usually selects the next decision variable among these variables; (2) they have tight relations with recently learnt clauses, because these learnt clauses contain them or have been obtained by resolving on them. Consequently, these variables have a high probability to participate in future conflicts to be resolved on during the conflict analysis.

Therefore, we define the ESA ordering to be the increasing order of the activities of the variables in recent conflicts. In practice, a decision heuristic in a state-of-the-art solver usually already quantifies these activities as a score. So, the ESA ordering is obtained by sorting the variables in the increasing order of their score induced by the decision heuristic. In the case where the solver switches between several decision heuristics, the score used to sort the variables is given by the current heuristic when BVE is called.

When there is no decision heuristic that can effectively quantify the participation of a variable in recent conflicts (e.g. in a pre-processing), the ESA ordering is the increasing order of the value $|\phi_x| \times |\phi_{\neg x}|$ in MiniSAT and $|\phi_x| \times |\phi_{\neg x}| + |\phi_x| + |\phi_{\neg x}|$ in Kissat.

We implement the ESA ordering in Algorithm 6, so that the low activity variables satisfying the two bounds *clauseNbGrowthLimit* and *resolventLengthLimit* are eliminated first, which may reinforce the high activity variables by increasing the number of clauses they occur in, possibly preventing them from being eliminated.

Note that if the number of clauses in which a high activity variable x occurs is not increased enough by the elimination of the low activity variables, x is still eliminated if it satisfies the bounds. So, the ESA ordering is a soft (or indirect) protection of the high activity variables, which is different from a hard protection (or direct) that, e.g., forbids the elimination of any variable with an activity higher than a threshold. Note that such a threshold probably should be instance-dependent and deserves future study.

4.2.2 Implementation of ESA in Cadical and Kissat Based Solvers

Cadical [23] is the CDCL solver that won the UNSAT Main track of the SAT Competition 2018. We use version 1.4.1, which is the specified version for the Hack track of the SAT Competition 2022. It switches between focused and stable modes. The decision heuristics VMTF (Variable Move-to-Front) [27] and VSIDS (Variable State Independent Decaying Sum) [145] are used to select decision literals in focused mode and stable mode, respectively. The VMTF heuristic gives the highest score to the variables participating in the most recent conflict, breaking ties using their (non-)participation in the second most recent conflict, the remaining ties being broken using their (non-)participation in the third most recent conflict, etc.... The VSIDS heuristic initializes the score of every variable to 0. Let inc be a real such that $0 < inc < 1$. The VSIDS score s of a variable participating in the conflict number i is updated by $s \leftarrow s + (1/inc)^i$.

The interval between two calls to BVE in Cadical is measured in the number of conflicts, which grows in $O(n)$, where n is the number of calls to BVE so far.

Cadical_ESA is Cadical except that BVE is called with the ESA ordering in the stable mode where VSIDS is used as decision heuristic. In other words, the variables are sorted in the increasing order of their VSIDS score in the BVEs called during the stable mode. In the focused mode where VMTF is used as decision heuristic, the original increasing ordering of $|\phi_x| \times |\phi_{\neg x}| + |\phi_x| + |\phi_{\neg x}|$ remains unchanged in BVE.

Kissat is originally an optimized C version of Cadical by the same author with some changes to data structures. As it is improved, its difference with Cadical becomes bigger and bigger, especially after other authors add new solving techniques to it. We implemented ESA in the in-processing BVE of Kissat_MAB (KM for short), the winner of the Main track in the SAT Competition 2021; Kissat_MAB_HyWalk [182] (KM_HW for short) and Kissat_Inc [47] (K_Inc for short), the best two solvers in the Main track of the SAT Competition 2022.

The three Kissat based solvers switch between focused mode, in which the VMTF decision heuristic is used, and stable mode, in which the solvers switch further between two decision heuristics VSIDS and CHB(Conflict History-based Branching) [128] to select decision literals, by means of MAB (Multi-Armed Bandit) borrowed from reinforcement learning [49], which is different from Cadical that uses only VSIDS in stable mode.

The CHB score $q(x)$ of a variable x is initialized to 0. After a unit propagation, $q(x)$ is updated, for each variable x assigned in this unit propagation, using the following Exponential Recency Weighted Average (ERWA) function, where $0 < \alpha < 1$ is a parameter and $r(x)$ is a reward:

$$q(x) \leftarrow (1-\alpha) \times q(x) + \alpha \times r(x) \tag{4.1}$$

Reward $r(x)$ is defined as follows:

$$r(x) = multiplier/(i - lastConflict(x) + 1) \tag{4.2}$$

where i is the number of conflicts so far, $lastConflict(x)$ is the last conflict number in which x participated, and $multiplier$ is 1 if the unit propagation that just terminated produces a conflict, otherwise it is 0.9.

The variables participating in recent conflicts are favored for two reasons: (1) $multiplier$ is greater for these variables, (2) $(i - lastConflict(x) + 1)$ is smaller.

The interval between two calls to BVE in the three Kissat based solvers is also measured in the number of conflicts, which grows in $O(n * log^2(n))$, where n is the number of calls to BVE so far. In addition, some planned BVEs are canceled if there are too many clauses. So, BVE is executed less frequently in the Kissat based solvers than in Cadical.

KM_ESA (KM_HW_ESA, K_Inc_ESA) is KM (KM_HW, K_Inc) except that BVE is called with the ESA ordering in the stable mode. When the BVE procedure is called at the moment VSIDS (CHB) is used as decision heuristic, the ESA ordering is the increasing order of the VSIDS (CHB) score.

As in Cadical, when the three Kissat based solvers are in the focused mode where VMTF is used as decision heuristic, the original variable ordering by the value $|\phi_x| \times |\phi_{\neg x}| + |\phi_x| + |\phi_{\neg x}|$ remains unchanged in BVE. In other words, ESA does not use the VMTF score to sort the variables in BVE.

Note that ESA cannot be implemented in MiniSAT based solvers such as Glucose [166] and MergeSat [138], because these solvers do not implement in-processing BVE.

4.3 Experimental Results

We first show the effectiveness of ESA on the set of the 5355 instances of the Anniversary track in the SAT Competition 2022, augmented with the 320 instances from the Main track in the SAT Competition 2022 that were not included in the Anniversary track. In other words, the benchmark set includes all instances used in the Application, Crafted, and Main tracks of all SAT Competitions up to 2022, a total of 5675 distinct instances, so that the results can give a global performance evaluation. All experiments were run with Intel Xeon CPUs E5-2680@2.40GHz under Linux with 31GB of memory. The time limit for solving each instance is 5000s for each solver, as in the SAT Competition.

Then we conduct experiments to analyze the behavior of ESA and the reasons of its effectiveness.

4.3.1 Effectiveness of the ESA Procedure

Table 4.1 gives the number of instances solved by Cadical, KM, KM_HW, K_Inc and their versions with ESA, the average time to solve these solved instances and the PAR2 score (computed as in SAT competitions as the average solving time of all the 5675 instances, the solving time of an unsolved instance counted as 2×timeout, i.e. 10000s). As can be seen, a solver with ESA consistently solves more instances than the base solver without ESA. Recall that the only difference between each solver with ESA and

Solver	Solved	Avg(s)	PAR2
Cadical	4222	443	2890
Cadical_ESA	4230	441	2875
Kissat_MAB_HyWalk	4252	388	2798
Kissat_MAB_HyWalk_ESA	4269	410	2786
Kissat_Inc	4282	410	2764
Kissat_Inc_ESA	4291	405	2745
Kissat_MAB	4283	415	2767
Kissat_MAB_ESA	4302	420	2738

Table 4.1: Results of Solvers with and without ESA

its baseline without ESA is the variable ordering used in BVE, which indicates the effectiveness and the robustness of ESA, as these results show the compatibility of ESA with different techniques implemented in KM, KM_HW, K_Inc and Cadical. It is worth noting that KM_ESA and Cadical_ESA participated in the SAT Competition 2022: KM_ESA won the Anniversary track, and Cadical_ESA solved more instances than all other variants of Cadical in the Anniversary track of the competition.

More specifically, KM_ESA solves 57(38) instances more(less) than its baseline KM, among which 2,2,2,3 and 4 more instances in families Ptn, tseitingrid, fphp, preimage and hcp, respectively; and 2 and 2 fewer instances in LABS and mchess, respectively. The difference does not exceed 1 on other families of instances. No significant difference is observed between sat and unsat instances.

Figure 4.3 gives the cactus plots of Cadical_ESA, KM_ESA and their baselines, showing that Cadical_ESA solves more instances than Cadical at most points, and KM_ESA solves more instances than KM with any timeout.

Note that KM and Cadical are already highly optimized and it is extremely hard to make them solve even one more instance. As a matter of fact, the best two solvers in the Main track of the SAT Competition 2022, namely, KM_HW and K_Inc, solve exactly the same number of instances in that track. Only a few seconds in solving time allowed to distinguish them in the main track. Our results show that ESA would allow them to win more clearly.

Table 4.2 further confirms the effectiveness of ESA by comparing KM and KM_ESA with the following variants:

Kissat_MAB_NE (KM_NE for short), it is KM with BVE disabled (i.e., no elimination);

Kissat_MAB_ESRall (KM_ESRall), it is KM, but variables are checked and eliminated in a random ordering in BVE both in focused and stable mode;

Kissat_MAB_ESN (KM_ESN), it is KM, but the variables are sorted for BVE in the natural integer ordering in stable mode and in the increasing order of $|F_x| \times |F_{\neg x}| + |F_x| + |F_{\neg x}|$ in focused mode;

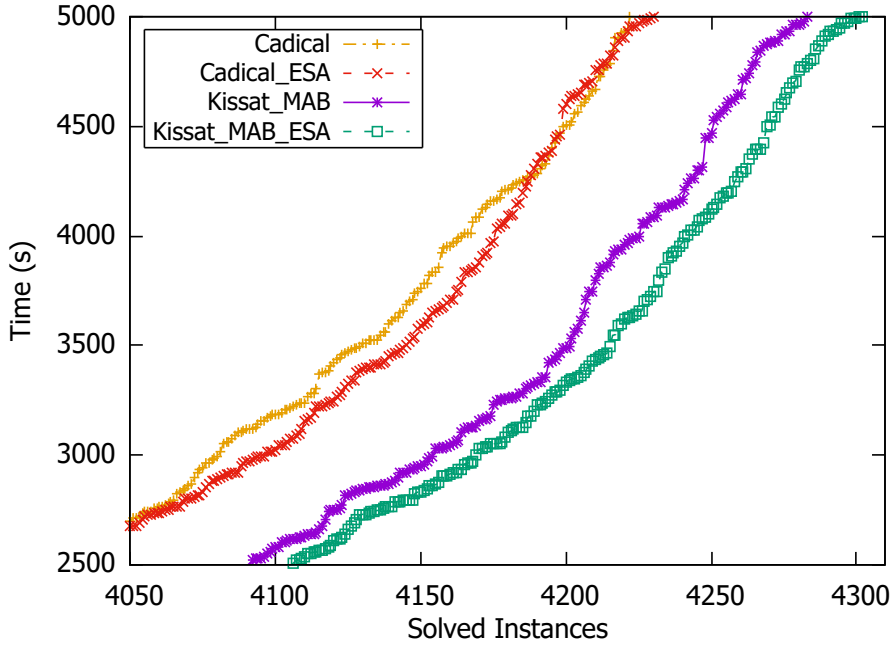


Figure 4.3: Cactus plots of different solvers. Each point (N, T) in a curve indicates the number N of instances solved within T secs by the corresponding solver. Recall that the frequence of BVE and decision heuristics are different in Cadical and KM.

Kissat_MAB_ESNall (KM_ESNall), it is KM, but the variables are sorted for BVE in the natural integer ordering in both focused and stable modes;

Kissat_MAB_Bin (KM_Bin), it is KM, but when a variable x is eliminated, let L^2_x ($L^2_{\neg x}$) denote the set of binary learnt clauses containing x ($\neg x$), add $L^2_x \otimes_x L^2_{\neg x}$ into L after removing all learnt clauses containing x or $\neg x$ from L. For more information please check Section 4.4.1.

Kissat_MAB_ESAfull (KM_ESAfull), it is KM_ESA, but when BVE is called in focused mode where VMTF is used as decision heuristic, the variables are sorted in the increasing ordering of their VMTF score in BVE instead of $|\phi_x| \times |\phi_{\neg x}| + |\phi_x| + |\phi_{\neg x}|$. In other words, KM_ESAfull uses all the three decision heuristics of KM for ESA.

A number of observations can be made from Table 4.2.

- In-processing BVE is effective, because KM solves 81 instances more than KM_-NE.
- The natural integer ordering is better than the random ordering, but is worse than the $|\phi_x| \times |\phi_{\neg x}| + |\phi_x| + |\phi_{\neg x}|$ ordering, because KM solves 6 (8) instances more than KM_ESN (KM_ESNall).

Solver	Solved	Avg(s)	PAR2
Kissat_MAB_NE	4202	460	2936
Kissat_MAB_ESRall	4262	415	2802
Kissat_MAB_ESNall	4275	406	2773
Kissat_MAB_ESN	4277	419	2780
Kissat_MAB	4283	415	2767
Kissat_MAB_ESAfull	4287	426	2768
Kissat_MAB_Bin	4296	423	2750
Kissat_MAB_ESA	4302	420	2738

Table 4.2: Results of Variant Solvers

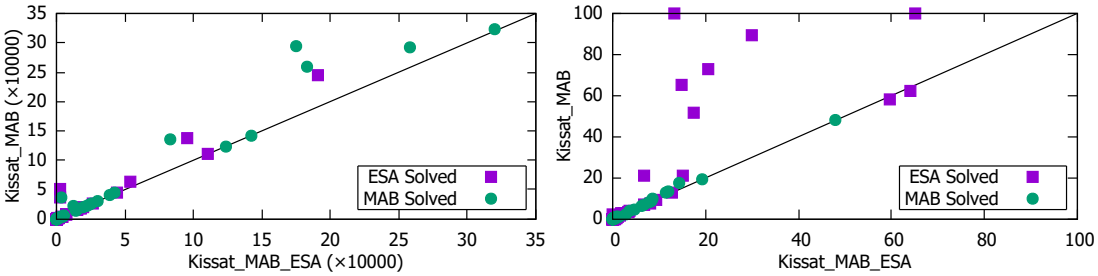


Figure 4.4: Scatter plot of the number of eliminated variables (left), and scatter plot of the number of learnt clauses lost per variable (right) by the 1st BVE of Kissat_MAB_ESA and Kissat_MAB.

- By keeping a part of resolution results so far, KM_Bin is significantly better than KM, but it does not reach the performance of KM_ESA. Indeed, the impact of an eliminated high activity variable x could be divided into two phases by the moment t it is eliminated. KM_Bin catches up a part of the impact of x before t, but loses its impact after t because it is eliminated. KM_ESA can have all its impact before and after t, because KM_ESA probably would not eliminate it.
- The VMTF score is not suitable to sort the variables in BVE, because KM_ESAfull is significantly worse than KM_ESA. It appears that not all effective decision heuristics in a CDCL solver are also effective to sort the variables in BVE, and only those decision heuristics reflecting accurately the activity of a variable in all recent conflicts are suitable to sort variables in BVE, which is not the case for VMTF.

4.3.2 Analyzing the Behaviour of ESA

In the Anniversary track of the SAT Competition 2022, KM_ESA solved 16 instances more than KM (named Kissat_MAB_UCB in the results) from a total of 5355 instances: KM_ESA solved 62 instances that KM did not solve (ESA Solved), and KM solved 46 instances that KM_ESA did not solve (MAB Solved). Figure 4.4 (left) compares the

number of variables eliminated by the first BVE of the two solvers when solving these 108 instances. Since the two solvers have exactly the same behavior and follow the same search trajectory until the first BVE, the unique reason for the difference in the first BVE is the variable ordering in BVE. It is clear that the ESA ordering eliminates fewer variables in the first BVE than the original variable ordering in MAB.

When a variable is eliminated in an in-processing, the learnt clauses containing this variable are lost. Figure 4.4 (right) compares the average number of lost learnt clauses per variable in KM_ESA and KM in the first BVE. KM_ESA loses fewer learnt clauses than KM when eliminating variables, which might partly explain the effectiveness of ESA.

Assume that KM_ESA (KM) calls k_1 (k_2) times BVE in total for solving an instance and $k = min(k_1, k_2)$. The above and below plots in Figure 4.5 compare respectively the average number of eliminated variables per BVE call, and the average number of lost learnt clauses per variable in these k BVE.

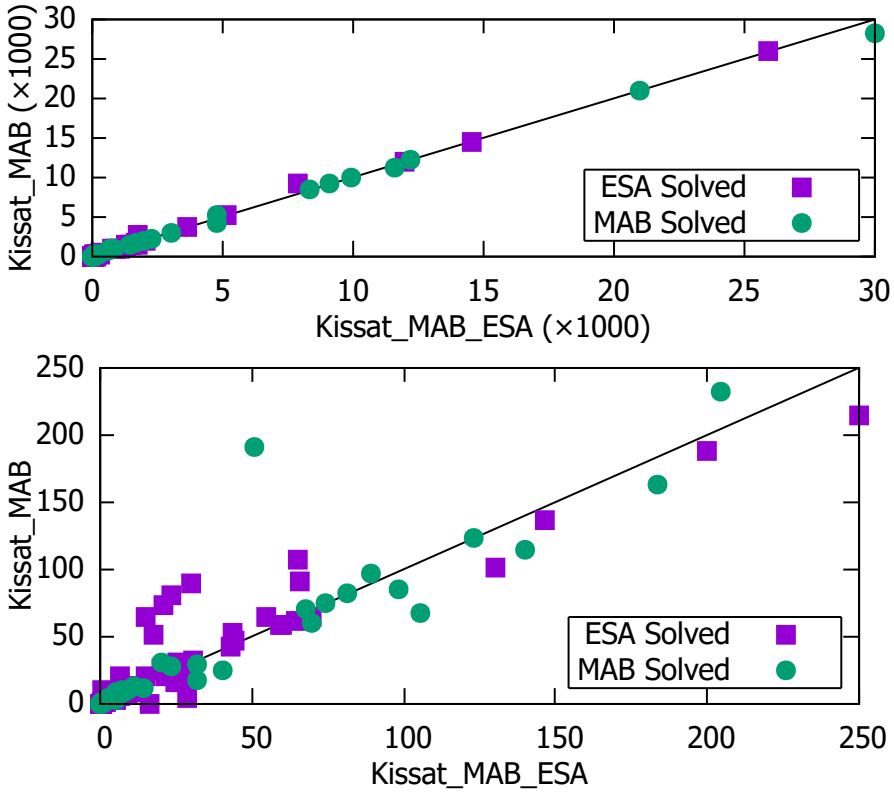


Figure 4.5: Scatter plot of the average number of eliminated variables per BVE call (above), and scatter plot of the average number of learnt clauses removed per variable among the first k BVE calls (below).

Although KM eliminates more variables in the first BVE, Figure 4.5 (above) shows that KM and KM_ESA eliminate roughly the same average number of variables per BVE call, meaning that KM_ESA eliminates more variables in the subsequent BVEs. This is reasonable for ESA, because it is probable that the important variables are not

yet fully identified before the first BVE, so that it is better that the first BVE does not eliminate too many variables, and leave more variables to eliminate in subsequent BVEs for which the important variables are better recognized with their activities. Figure 4.5 (below), together with Figure 4.4 (right) and the good results of KM_Bin, suggests that one reason for which KM fails to solve an ESA solved instance might be that KM lost too many learnt clauses per eliminated variable.

4.4 Other Implementations

The improvement of ESA on the Kissat-based solvers inspired us that the heuristic score of variables can also be used to guide other simplification functions. Besides, the Figure 4.5 (below), together with Figure 4.4 (right) also suggested that keeping the learnt clauses when eliminating variables may help solve. We introduce the two main implementations below.

4.4.1 Keep Binary Learnt Clauses after Variable elimination

In the inprocessing bounded variable elimination progress, Kissat [26] and other Kissat-based solvers only keep the resolvents of original clauses (or irredundant clauses in Kissat) containing literal x or $\neg x$ when they eliminate variable x. As for those learnt clauses (or redundant clauses in Kissat) containing x or $\neg x$, they just delete them directly, which results in a loss of information contained in those learnt clauses.

After eliminating x, clauses containing it cannot be used anymore in the subsequent search, so we can't keep those learnt clauses directly, but keeping the resolvents of those learnt clauses seems a good way to preserve those learnt information. However, the number of learned clauses containing x may be several orders of magnitude larger than the number of original clauses containing x, so that keeping all the resolvents will have a huge memory cost and the benefits are far outweighed by the performance loss. Because learnt clauses are redundant, it's not necessary for us to keep all their resolvents, but those with high quality. A clause with small LBD or length is often considered a high-quality clause. In our solver Kissat_MAB_Binary, we only keep the resolvents of binary learnt clauses relating to those eliminated variables. The experimental results are shown in Table 4.2.

4.4.2 Variable Heuristic in Other Simplification Functions

We developped another SAT solver AMSAT (Amiens Marseille SAT solver), which is based on lstech_maple [181], and is the best MiniSAT [67] based solver in SAT competition 2021. We added the following improvements, essentially based on variable heuristic score and inspired by ESA [122].

Failed / Implied / Equivalent Literal Detection AMSAT detects failed / implied / equivalent literals following the Algorithm 7 in Chapter **??**.

At level 0, the detected failed, implied, and equivalent literals are global and do not depend on any partial assignment. In other words, if p is failed, then $\neg p$ is satisfied directly. An implied literal q is satisfied directly. The equivalent literals are not substituted immediately, because AMSAT does not maintain a variable-to-clause structure permanently. Each equivalent literal will be substituted by an arbitrarily chosen literal in its equivalent class later upon a call to the local search solver CCAnr, together with clause subsumption to facilitate the work of CCAnr.

Since equivalent literals are not substituted immediately, we have to deal with the non-symmetric property of unit propagation when there are clauses containing more than 2 literals. Namely, when unit propagation deduces q from p, it does not necessarily deduce $\neg p$ from $\neg q$, although $p \rightarrow q$ is equivalent to $\neg q \rightarrow \neg p$. For two equivalent literals p and q, a binary clause $\neg p \vee \neg q$ is added if unit propagation fails to deduce q from p. Note that these added binary clauses are removed when equivalent literals are substituted.

At levels other than 0, the detected failed, implied and equivalent literals depend on a partial assignment. For simplicity reason, AMSAT only detects failed and implied literals at levels other than 0, and learns an asserting clause for each failed literal and each implied literal. It is easy to derive an asserting clause for a failed literal using the usual conflict analysis. For an implied literal q, let $p \rightarrow q$ and $\neg p \rightarrow q$, a clause $cl_1 \vee \neg p \vee q$ ($cl_2 \vee p \vee q$) can be derived from the reason clause of q in the implication graph propagating p ($\neg p$), where cl_1 and cl_2 are sub-clauses contains literals falsified before p or $\neg p$ and having a path to q in the implication graph. Then, the resolvent $cl_1 \vee cl_2 \vee q$ of the two clauses asserts q.

Inspired by ESA, AMSAT selects the variables in the decreasing order of their current activity (VSIDS or LRB) and detects x and $\neg x$ respectively. Let $limit$ be a parameter fixed to 100 at level 0, and 10 at other level, the detection stops if no failed, implied or equivalent literal is found for $limit$ variables since the last failed, implied or equivalent literal is found. In order to further limit the overhead, these detections are only executed at levels 0, 1 and 2. At level 0, failed, implied and equivalent literal detection is also executed in addition before each clause vivification.

Inprocessing Bounded Variable Elimination There is no inprocessing variable elimination in previous MiniSAT-based solvers, and AMSAT adds this technology. During solving, when 2% of variables are fixed, eliminated or substituted at level 0, the inprocessing bounded variable elimination is executed. A limited part of resolvents of learnt clauses with small LBD and size containing the eliminated variables is also kept after variable elimination. All obtained resolvents from variable elimination are vivified.

Deleting Learnt Clauses by Variable Activity In MiniSAT-based solvers, the indicator for deleting learnt clauses is their activity, which roughly reflects the fre-

quencies of these learnt clauses being involved in conflicts. Let LOCAL be the subset containing clauses with LBD greater than 6 or not involved in any conflict since 30000 conflicts. The half of lower activity learnt clauses in LOCAL is removed every 15000 conflicts. AMSAT does not use learnt clause activity but variable activity to select the clauses to remove. Concretely, let $act1$ and $act2$ be the smallest and the second smallest variable activities in a learnt clause, AMSAT sort the clauses in LOCAL in the increasing of their $(act1 \times w + act2)/(LBD * LBD)$, where w is a parameter fixed to 1000. The first half is removed every 15000 conflicts.

In addition to the above ESA-inspired technologies, we have also made the following improvements in AMSAT.

Variable Mapping and Space Compression In SAT solving, if a variable becomes inactive (fixed at level 0, eliminated, etc.), we no longer need the data related to it. And if the number of inactive variables is large, there are many holes in the array used to record variable/literal data, resulting in increased traversal costs. MiniSAT-based solvers do not deal with this problem. AMSAT compresses the space by removing the holes using variable mapping. Note that the final solution generation for satisfiable instances and the unsat proof generation for unsatisfiable instances should be adapted accordingly.

Reducing the Frequency of Clauses Garbage Collection . The clauses garbage collection machanism in AMSAT is the same as in MiniSAT, it is triggered When the memory space occupied by deleted clauses is more than 20% of the total space. In MiniSAT, this time consuming process will first allocate for a new memory, copy all clauses into the new memory, and release the deleted clauses at the same time. To reduce the cost of garbage collection, we raise the threshold for executing it: when clauses number is small, solvers do garbage collection after 40% clauses are deleted, this threshold decreases as the clauses number increases until 10%. We implemented this modification in solver AMSAT_RGC [****].

4.5 Solvers at SAT Competitions

We submitted our solver Kissat_MAB_ESA and Cadical_ESA to the SAT Competition 2022. The solver won the anniversary main track of the competition and placed 2_{nd} in the anniversary unsat track. The summary of the results in the anniversary main track are reported in Figure 4.6.

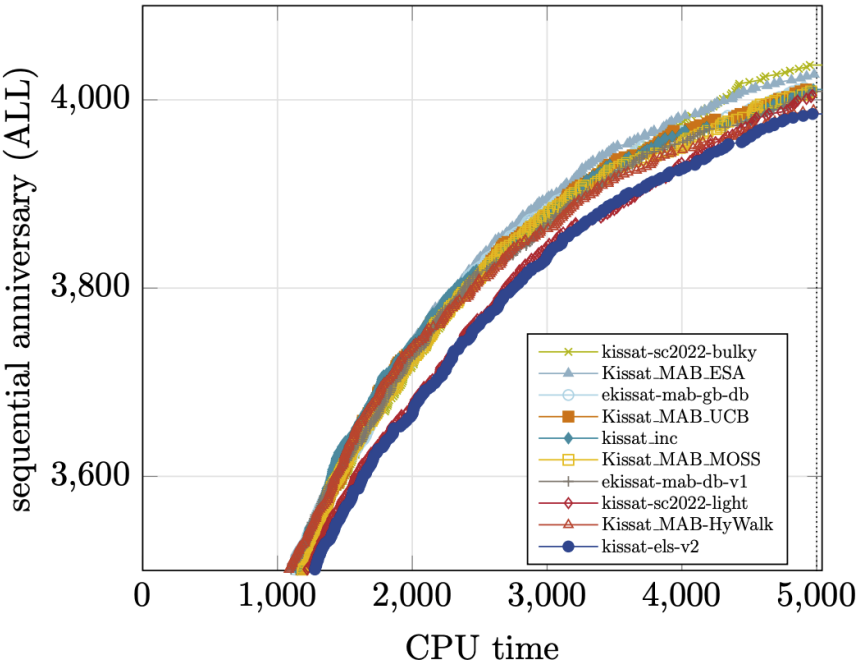


Figure 4.6: Anniversary main track results of the SAT Competition 2022

4.6 Conclusion

In this chapter, we first analyzed the limitations of current in-processing bounded variable elimination: (1) when a variable is eliminated, all learnt clauses containing it are also removed, so that a part of the results in the CDCL search so far is lost; (2) an eliminated variable cannot be resolved on anymore in the subsequent CDCL search, hurting the power of resolution in CDCL. Thus, we propose a new variable ordering, called ESA (variable Elimination Scheduled by Activity), to remedy the above two drawbacks. In practice, the ESA ordering is obtained by using the decision heuristic of the solver that is able to reflect accurately the participation of the variables in recent conflicts. Consequently, the low activity variables satisfying the bounds are eliminated first in BVE, providing a soft protection to the high activity variables.

We implemented the ESA ordering in Cadical and three winning solvers based on Kissat which call in-processing BVE differently and implement different decision heuristics and other techniques. Experimental results show that ESA consistently allows these solvers to solve more instances in the set of all the 5675 Crafted, Application and Main track instances of all SAT Competitions up to 2022, demonstrating the effectiveness and robustness of ESA. An empirical analysis is carried out to analyze the reasons of the effectiveness and robustness of ESA.

Due to the good performance of ESA, the principle behind it were also used to improve other simplification techniques in SAT solvers, including failed/implied/equivalent literals detection and learnt clauses eliminations. These improvements were implemented on a MiniSAT based solver and enhanced its perfor-

mance.

ESA has also inspired us to realize that in SAT solving, the processes of search and simplification are not necessarily independent of each other. The exchange of information and collaboration between them could further enhance the solver's performance. As future work, it would be interesting to further explore the execution processes of search and simplification to identify other relationship between them.

5 Improving the Lower Bound in Branch-and-Bound Algorithms for MaxSAT

Sommaire

5.1 Motivation 84
5.2 Lookahead with Unlocking Mechanism for Unweighted MaxSAT 85
5.3 Lookahead with Unlocking Mechanism for Weighted MaxSAT 86
5.4 Experimental Evaluation 90
5.4.1 Experimental settings 90
5.4.1.1 Benchmarks 90
5.4.1.2 Compared solvers 90
5.4.1.3 Experimental environment 91
5.4.2 Impact of the unlocking mechanism 91
5.4.3 Empirical analysis 91
5.5 Conclusions and Future Work 93

5.1 Motivation

The core detection method in modern Branch and Bound solvers have two limitations, which we illustrate by an example. In Example 5.1, with different soft literals assumptions we get different cores, and both of them have literal s_1. To keep the disjointness between different cores, c_1 and c_2 cannot exist as the same time. The two limitations are: (1)The literals locked in one core cannot be used in another core. (2)Only one guaranteed falsified soft literal in each core.

Example 5.1 *Given an unweighted MaxSAT formula* $\phi = \{\{\neg s_1, \neg x_1\}, \{\neg s_2, \neg x_2\}, \{\neg s_3, \neg x_3\}, \{x_1, x_2, x_3\}, \{\neg s_1, x_4\}, \{\neg s_4, x_5\}, \{\neg x_4, \neg x_5, \neg s_5\}, \{\neg s_4, \neg s_2\}\}$, *it includes 8 hard clauses with 5 soft literals and 5 literals inside. As in Figure 5.1, by propagating* s_1, s_2, s_3 *we get a conflict, so that we find core* $c_1 = (s_1, s_2, s_3)$ *and the new lower bound* $LB = 1$. *Or, we can propagate* s_1, s_4, s_5 *and get another core* $c_2 = (s_1, s_4, s_5)$ *with* LB *is also* 1, *as in Figure 5.2.*

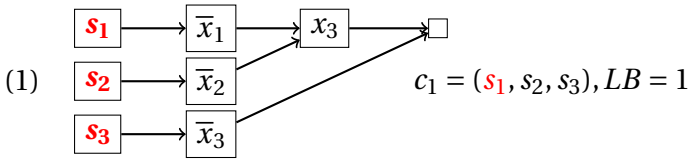


Figure 5.1: The process of normal core detection, soft literals s_1, s_2, s_3 are propagated

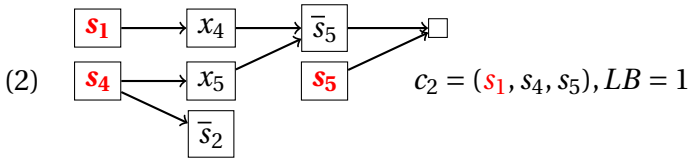


Figure 5.2: The process of normal core detection, soft literals s_1, s_4, s_5 are propagated

The two limitations actually hurts the effectiveness of lower bound estimation, since the soft literals that already appear in the cores are more likely to cause conflicts, and locking them would harm subsequent core detection. In this chapter, we propose a core unlocking mechanism, which aims to detect better cores and ensure efficient pruning.

5.2 Lookahead with Unlocking Mechanism for Unweighted MaxSAT

Following the MaxCDCL approach, we replace each soft clause cl by a soft literal s and add hard clauses encoding $s \leftrightarrow cl$. This allows us to deal with cores of soft literals instead of cores of soft clauses. For each core c, $w(c)$ indicates that c contains at least $w(c)$ falsified soft literals for any complete extension of the partial assignment α. A core c is *locked* if $w(c) > 0$. An unassigned soft literal is $locked$ if it belongs to a locked core. Our *unlocking mechanism* consists in unlocking the soft literals in c when there are $w(c)$ falsified soft literals in c, allowing its unassigned soft literals to be reused for detecting other cores.

Alg. 11 depicts the process of lookahead to detect a set of cores using the unlocking mechanism, given a set of hard clauses H and a set of unassigned soft literals S. It propagates the soft literals in S one by one as assumptions until an empty hard clause cl is produced or a literal s' in S is falsified. Then, it retraces the implication graph by collecting all propagated assumptions starting a path to cl or s' to form a core. The main difference with MaxCDCL is the unlocking operations from line 6 to line 11 which re-insert into S soft literals previously locked in a core c. These soft literals are unlocked because there are $w(c)$ literals in c falsified during UP ($w(c)$ becomes 0). Since there are more soft literals in S after these insertions, detecting new cores becomes easier.

Example 5.2 *Let c_1={s_1, s_2, s_3} be a core already detected, and $w(c_1) = 1$. Its soft literals s_1, s_2 and s_3 are locked. Alg. 11 propagates now s_4 as an assumption, falsifying s_1, so that $w(c_1)$ is decreased to 0, and s_2 and s_3 are unlocked. It then propagates s_2 as an assumption, which falsifies s_5, an unassigned soft literal in S. So, a new core c_2={s_4, s_2, s_5} is identified, in which s_2 belongs to c_1. So, c_2 is updated to the union with c_1: {s_1, s_2, s_3, s_4, s_5} in line 17, and $w(c_2)$ is set to 2 in line 18 to lock all its literals. Note that c_1 should be removed after finalizing c_2.*

After detecting a new core c_3={s_6, s_7} with $w(c_3) = 1$, Alg. 11 propagates s_8 as an assumption, which falsifies s_5 and s_6, unlocking s_7, and $w(c_2)$ is decreased to 1. Propagating s_7 as an assumption falsifies s_4, unlocking s_1, s_2 and s_3 because $w(c_2)$ is 0 now. Continuing UP, s_3 is falsified. Thus, a new core c_4={s_8, s_7, s_3} is detected. Then, c_4 is modified in line 17 to the union with c_2 and c_3, $w(c_4)$ is set to 4 in line 18, and c_2 and c_3 are removed in line 20.

If no additional cores can be detected, Alg. 11 will return 4, which is the sum of the locks of all remaining cores.

Note that without unlocking, s_2 could not be propagated as an assumption and c_2={s_4, s_2, s_5} would not be detected because s_2 remains locked in c_1. Similarly, c_4={s_8, s_7, s_3} would not be detected, because s_3 is also locked in c_1. Consequently, LB would be 2 without unlocking.

Proposition 5.1 establishes the soundness of Alg. 11.

Proposition 5.1 *Any complete assignment extended from α falsifies at least $w(d)$ soft literals in the core d identified at line 17 of Alg. 11. After removing cores in C in line 20, d is disjoint with the remaining cores, and the returned LB is equal to $\sum_{c \in cores} w(c)$.*

Proof 5.1 *By induction. When $w(d)$=1, by construction, d does not contain any soft literal of other cores, and at least one soft literal must be falsified by any complete assignment.*

Assume the proposition holds for $w(d)<k$. When $w(d)=k$, for any c in C obtained in line 16, we have $w(c) < k$. So, these cores c are pairwise disjoint, and an assignment falsifies at least $\sum_{c \in C} w(c)$ soft literals. Let d_0 denote the subset of soft literals of d that do not belong to any core in C. If an assignment falsifies a soft literal in d_0, it falsifies at least $1 + \sum_{c \in C} w(c)$ soft literals in d, because $d_0 \cap c = \emptyset$ for any $c \in C$. If an assignment satisfies all soft literals in d_0, UP falsifies at least $w(c)$ literals in each c in C except one c in which UP falsifies $w(c)+1$ soft literals. Note that d is disjoint with cores not in C and cores in C are removed after finalizing d.

Finally, each new core increases LB by 1 in line 19, because $\sum_{c \in C} w(c)$ was already counted in LB. So, the final LB is equal to $\sum_{c \in cores} w(c)$.

5.3 Lookahead with Unlocking Mechanism for Weighted MaxSAT

Unfortunately, the beneficial property established by Proposition 5.1 does not hold for weighted MaxSAT. So, we cannot guarantee the number of falsified soft literals in

weighted MaxSAT as we can in unweighted MaxSAT. The reason is that, in unweighted MaxSAT, a soft literal belongs to at most one core, whereas in weighted MaxSAT, a soft literal can belong to multiple cores. Consequently, a naive extension of Alg. 11 could result in a less accurate LB.

Example 5.3 *Let s_1, s_2, s_3, s_4, s_5, s_6 and s_7 be soft literals with weights 1, 1, 4, 3, 5, 6 and 7, respectively. Using Alg. 11, we first identify a core c_1={s_1, s_2, s_3} and $w(c_1)$=1. The remaining weight $rw(s_3)$=3. Next, we identify another core c_2={s_3, s_4, s_5} and $w(c_2)$=3, resulting in an LB of* 4*. Subsequently, we propagate s_6, which falsifies s_3 and unlocks soft literals in c_1 and c_2. This allows us to propagate s_1 and s_4, which falsifies s_7. Finally, we obtain c_3={s_1, s_2, s_3, s_4, s_5, s_6, s_7} and remove c_1 and c_2 using a naive extension of Alg. 11, because we have propagated s_1 in c_1 and s_4 in c_2 to obtain c_3. The problem is that s_3 belongs to both c_1 and c_2. Thus, we cannot guarantee three falsified soft literals in c_3; we can only guarantee two falsified soft literals. Since s_1 and s_2 have the smallest weight of 1 in c_3, this results in an LB of 2, instead of the previous LB of 4 before detecting c_3.*

So, extending the unlocking mechanism to weighted MaxSAT is not trivial. The main difficulty lies in how to compute the weight $w(c)$ of a newly detected core c, the part of weight of a soft literal s locked in c to decide $w(c)$, denoted as $wlock(s,c)$, and its remaining weight $rw(s)$ not locked in any core, when c includes soft literals already locked in previous cores. Example 5.4 suggests our approach to compute $w(c)$, $wlock(s,c)$ and $rw(s)$, based on the following unlocking mechanism: a soft literal s such that $rw(s)>0$ can be freely propagated as an assumption to detect a new core. If $rw(s)=0$, s is locked, but if $\sum_{s'\in c, value(s')=0} wlock(s',c)$ reaches $w(c)$ during UP, then s is unlocked with available weight $wlock(s,c)$, and can also be freely propagated as an assumption. In this case, we say that c is *unlocked* and c *unlocks* s. For convenience, $wlock(s,c)=0$ if s is not in c.

Example 5.4 *Refer to Example 5.3. Once c_1={s_1, s_2, s_3} and c_2={s_3, s_4, s_5} are detected, $w(c_1)$=1, $w(c_2)$=3, $wlock(s_1,c_1)$=1, $wlock(s_3,c_1)$=1, $wlock(s_3,c_2)$=3, $wlock(s_4,c_2)$=3, $rw(s_1)$=$rw(s_2)$=$rw(s_3)$=$rw(s_4)$=0, $rw(s_5)$=2, $rw(s_6)$=6, $rw(s_7)$=7.*

Propagating s_6 as an assumption falsifies s_3 and unlocks c_1 and c_2. Then s_1 and s_4 can be propagated as assumption, falsifying s_7 and giving a new core c_3. So, s_6, s_1 and s_4 are propagated assumptions starting a path to $\neg s_7$ in the implication graph G, involving cores c_1 and c_2, and c_3={s_6, s_1, s_4, s_7}$\cup c_1 \cup c_2$. We now focus on s_6, s_1, s_4, s_7 in the ordering they are assigned a value to compute $w(c_3)$. Indeed, any complete assignment extending α falls into one of the following four cases: (1) s_6=0, (2) s_6=1 and s_1=0, (3) s_6=s_1=1 and s_4=0, and (4) s_6=s_1=s_4=1 and s_7=0. We analyze the total weight falsified in each of these four cases and set $w(c_3)$ to the minimum of them.

In case (1), the falsified weight is $rw(s_6)+w(c_1)+w(c_2)$ because c_1 and c_2 are disjoint. In the other cases, c_1 and c_2 are unlocked, meaning that the falsified weight is already $w(c_1)+w(c_2)$, unlocking s_1 and s_4 and making $wlock(s_1,c_1)$ and $wlock(s_4,c_2)$ available. The additional falsified weight in the three cases is $rw(s_1)+wlock(s_1,c_1)$=1,

$rw(s_4) + wlock(s_4, c_2)$=3 and $rw(s_7)$=7, respectively. Note that $w(c_1) + w(c_2)$ is common in the four cases. Consequently, $w(c_3)$ is equal to $w(c_1) + w(c_2) + min(6,1,3,7)$=5.

Now we compute $wlock(s, c_3)$ for each s in c_3, which is the minimum weight of s contributing to compute $w(c_3)$=5. First, s_2, s_3 and s_5 do not participate in the computation of $w(c_3)$ apart from that they belong to c_1 or c_2. So, $wlock(s, c_3)$=$wlock(s, c_1)$+$wlock(s, c_2)$ for $s \in \{s_2, s_3, s_5\}$. For $s \in \{s_6, s_1, s_4, s_7\}$, $wlock(s, c_3)$ must not be less than the weight already locked in c_1 and c_2, and not less than $min(6,1,3,7)$, the increase of $w(c_3)$ w.r.t. $w(c_1)$+$w(c_2)$. For example, to keep $w(c_3)$=5 unchanged, the contribution of s_6 to c_3 must not be less than 1. Formally, $wlock(s, c_3)$=$max(wlock(s, c_1)$+$wlock(s, c_2)$, $min(6,1,3,7))$. So, $wlock(s_6, c_3)$=1, $wlock(s_1, c_3)$=1, $wlock(s_4, c_3)$=3, $wlock(s_7, c_3)$=1. Under these conditions, $w(c_3)$=5. Finally, we compute the remaining weight $rw(s)$ for each $s \in c_3$. Again, $rw(s)$ does not change for $s \in \{s_2, s_3, s_5\}$. But $rw(s_6)$ and $rw(s_7)$ should be reduced by their weight newly locked in c_3. So, $rw(s_6)$=5 and $rw(s_7)$=6, and $rw(s_1)$ and $rw(s_4)$ remain 0, because they contribute to c_3 by their weight locked in c_1 or c_2.

Note that the weight of s_5, s_6 or s_7 locked in c_3 is substantially smaller than $w(c_3)$, leaving their remaining weight to detect other cores.

We now introduce Alg. 12, which, given a set of hard clauses H, a set of weighted soft literals S and a partial assignment α, detects cores as illustrated in Example 5.4. The main *while* loop can be divided into two parts. The first part (line 5 to line 15) repeatedly picks and propagates a soft literal from S. The distinguishing feature of this part is the unlocking of cores and their locked soft literals u (i.e., u with $rw(u)$=0) when the weight of these cores is decreased to 0 (lines 9 to 14). Note that the cases $rw(s') > 0$ or $wlock(s', c) > w(c)$ indicate the detection of a new core, and are treated in line 7. This situation arises because the total falsified weight during UP is greater than the total weight of the unlocked cores. Since $w(c)$ is changing, we use $ow(c)$ to denote the original weight of c computed in line 29.

The second part (from line 16 to line 35) is executed when a new core is detected. The propagated assumptions starting a path to the empty hard clause cl or the falsified literals fs in the implication graph G are collected into d, in which fs is also added, and the cores that unlocked a propagated assumption in d are collected in C. This part mainly computes $w(d)$, $wlock(s, d)$ and $rw(s)$ for each soft literal s in d.

Let $s_1, \ldots, s_k$ be assumptions in d propagated in this ordering. The assignments extended from α can be partitioned into the following k+1 cases: (1) s_1=0; (i)=(2)$\cdots$(k) s_1=$\cdots$=s_{i-1}=1, s_i=0; (k+1) s_1=$\cdots$=s_k=1. We examine the total guaranteed falsified weight in each of these (k+1) cases, because $w(d)$ is equal to the minimum of them.

In case (1), s_1 must not be an unlocked literal and $rw(s_1)$>0 because no core is unlocked before s_1 is propagated, and the total guaranteed falsified weight in the cores of C is $\sum_{c \in C} ow(c)$. So, the total guaranteed falsified weight is $rw(s_1) + \sum_{c \in C} ow(c)$.

In cases (2)$\cdots$(k), some cores in C could be unlocked because the propagation of s_1, ..., s_{i-1} could falsify some locked literals other than s_i in these cores, giving falsified weight $\sum_{c \in C, c \prec s_i} ow(c)$, where $c \prec s_i$ means core c was unlocked before s_i is propagated. Since the unlocking of these cores is independent of s_i, the weight of s_i

locked in these cores c ($wlock(s_i, c)$) becomes available. The total guaranteed falsified weight is thus $rw(s_i) + \sum_{c \in C, c \prec s} wlock(s_i, c) + \sum_{c \in C} ow(c)$.

Note that if c is not unlocked by the propagation of $s_1, \ldots, s_{i-1}$, $wlock(s_i, c)$ can be counted in $\sum_{c \in C, c \not\prec s_i} ow(c)$ in cases $(2) \cdots (k)$. We set the condition $c \prec s$ in the total falsified weight to exclude this situation.

Example 5.5 *Let c_1={s_1, s_2, s_3}, c_2={s_3, s_4, s_5} be two cores, where the weight of each soft literal is 1 except for s_3, which has a weight of 2 ($w(s_3)$=2). Thus, $ow(c_1)$=$ow(c_2)$=1, and $wlock(s_3, c_1)$=$wlock(s_3, c_2)$=1. When s_6 is propagated as an assumption, s_1 is falsified, unlocking s_2 and s_3 in c_1. Propagating s_2 and then s_3 as assumptions, falsifies s_4, unlocking s_5. Continuing UP, s_5 is falsified, producing a new core with propagated assumptions s_6, s_2 and s_3. Note that c_2 is unlocked after s_3 is propagated.*

For the assignments including s_6=s_2=1 and s_3=0, the falsified weight is $ow(c_1)$+$ow(c_2)$+$wlock(s_3, c_1)$. We can include $wlock(s_3, c_1)$ because $ow(c_1)$ is given by $wlock(s_1, c_1)$. However, we cannot include $wlock(s_3, c_2)$ because it can be $ow(c_2)$ which is already included.

In case $(k+1)$, we have two sub-cases: (i) an empty clause cl is produced, then s_1=$\cdots$=s_k=1 is not a feasible assignment and can be ignored; (ii) we have a falsified soft literal fs such that $rw(fs) > 0$ or $\exists c \in C$ such that $wlock(fs, c) > w(c)$. If fs is not in any core, the total guaranteed falsified weight is equal to $rw(fs) + \sum_{c \in C} ow(c)$. Otherwise, the cores containing fs are unlocked before fs is assigned, except eventually those c with $wlock(fs, c) > w(c)$. The falsified weights in these cores in addition to $\sum_{c \in C} ow(c)$ due to fs=0 can be written as $\sum_{c \in C}(wlock(fs, c) - w(c))$, where $w(c)$=0 if c is unlocked and $wlock(fs, c)$=0 if $fs \notin c$.

In all the $k+1$ cases, the total guaranteed falsified weight can be written as $rw(s) + \sum_{c \in C, c \prec s}(wlock(s, c) - w(c)) + \sum_{c \in C} ow(c)$ by extending the definition $c \prec s$ to include the case s is fs. Note that $\sum_{c \in C, c \prec s}(wlock(s, c) - w(c))$ is 0 in case (1), and is $\sum_{c \in C, c \prec s} wlock(s, c)$ in cases $(2) \cdots (k)$ because $w(c)$=0. The first two terms $rw(s) + \sum_{c \in C, c \prec s}(wlock(s, c) - w(c))$ in the guaranteed falsified weight is called *available weight* of s when s is propagated or s is fs, and is denoted as $aw(s)$.

In summary, $w(d)$ is the minimum guaranteed falsified weight among the $k+1$ cases, which can be written as $min_{s \in d} aw(s) + \sum_{c \in c} wo(c)$ in line 29.

Note that $w(d) > \sum_{c \in C} ow(c)$. The increase is $min_{s \in d} aw(s)$ and depends only on the propagated assumptions and fs. So, the remaining weight and the weight locked in d of other literals are not changed. For a propagated assumption s in d, $wlock(s, d)$ is the minimum value that is not smaller than $min_{s \in d} aw(s)$ and its total weight already locked in cores in C. So, $wlock(s, d)$ is equal to $max(\sum_{c \in C} wlock(s, c), min_{s \in d} aw(s))$ (line 25). Its remaining weight $rw(s)$ is computed accordingly (line 26).

From the above analysis, we obtain Proposition 5.2.

Proposition 5.2 *Let d be the core identified in line 30 in Alg. 12. Any complete extension of α falsifies at least weight $w(d)$ in d, and at least weight LB in ϕ.*

Remark. For unweighted MaxSAT, Alg. 12 behaves as Alg. 11, in which every thing is much simpler. We introduced Alg. 11 before Alg. 12 to facilitate the understanding of Alg. 12.

5.4 Experimental Evaluation

In this section, we evaluate and analyze the performance of the unlocking mechanism. To this end, we implemented Alg. 11 into MaxCDCL for unweighted MaxSAT, and Alg. 12 into WMaxCDCL for weighted MaxSAT. Since WMaxCDCL frequently splits the weight of a soft literal into locked and remaining weight, which is in complex interaction with the unlocking mechanism, we implemented Alg. 12 in two stages: first, with the unlocking mechanism disabled (i.e., lines 9 to 14). If the first stage is not successful, i.e., if LB<UB after the first stage, Alg. 12 is re-run with the unlocking mechanism enabled to detect additional cores by unlocking existing cores and their locked soft literals.

5.4.1 Experimental settings

5.4.1.1 Benchmarks

We use all instances of the weighted and unweighted exact tracks of the MaxSAT evaluation (MSE) from 2019 to 2023. We divide these 1962 unweighted instances and 2019 weighted instances into the subsets below.

(W)MS19∩20: subset of (weighted) instances used both in MSE19 and MSE20, denoted as 19∩20.

(W)MS19–(W)MS23: (W)MSy, denoted as y, where y ranges from 19 to 23, represents the subset of (weighted) instances used in MSEy but not in (W)MS19∩20 and MSEz for $z < y$.

5.4.1.2 Compared solvers

(W)MaxCDCL: the baseline solvers that can be found in the website of MSE23, listed as (W)M in the tables.

(W)MaxCDCL+Unlock: it is (W)MaxCDCL in which Alg. 11 (12) is implemented, listed as (W)MU. See supplement of this submission for sources codes.

WMaxCDCL-S6-HS12 [**wmaxcdcl2023**], **MaxCDCL-S6-HS9** [**maxcdcl2023**], **EvalMaxSAT-SCIP** [**evalmaxsat2023**] and **CASHWMAXSAT-CorePlus** [**cash2023**]**:** The best three solvers in the weighted and unweighted exact tracks of MSE23. All are portfolios that incorporate the integer programming solver SCIP [**bestuzheva2021scip**], and are listed as W23, M23, Eval and Cash, respectively. Specifically, M23 (W23) runs SCIP for 600s, then MaxHS

[**maxhs2022**] for 900s (1200s), and finally, (W)MaxCDCL2023 for 2100s (1800s) to solve an instance.

MaxSAT solving has reached a high level of maturity, making further improvements increasingly difficult. In fact, the winner of the unweighted (weighted) exact track of MSE23 solved only 3 (4) more instances than the second-place solver, out of a total of 572 (558) instances.

5.4.1.3 Experimental environment

All experiments ran on Intel Xeon CPUs E5-2680@2.40GHz under Linux with 31GB of memory. The timeout is set to 3600s per instance, as in MSE.

5.4.2 Impact of the unlocking mechanism

Table 5.1 presents the results for (W)MaxCDCL with and without the unlocking mechanism. The data show that the unlocking mechanism consistently enables solving more instances. If duplicated instances had not been removed, the difference between the two versions would be even greater. For instance, with the results of (W)MS19∩20 added, the unlocking mechanism increases the number of solved instances by 9 (6) in MSE19, and by 8 (13) in MSE20. A closer analysis reveals that there are 8 (3) instances solved by (W)MaxCDCL but not by (W)MaxCDCL+Unlock, and 38 (33) instances solved by (W)MaxCDCL+Unlock but not by (W)MaxDCL.

Figure 5.3 presents cactus plots showing the number of instances solved within x seconds. Across all values of x, (W)MaxCDCL+Unlock solves more instances than the baseline solver.

We replace (W)MaxCDCL in the portfolio MaxCDCL-S6-HS9 (WMaxCDCL-S6-HS12) by (W)MaxCDCL+Unlock and compare the resulting solver M23U (W23U) with MaxCDCL-S6-HS9 (WMaxCDCL-S6-HS12), Eval and Cash, which are the best three solvers in the unweighted and weighted exact tracks of MSE23. Tables 2 and 3 show that although a good part of the instances solved by (W)MaxCDCL+Unlock but not by (W)MaxCDCL are also solved by the two other components of MaxCDCL-S6-HS9 (WMaxCDCL-S6-HS12), M23U (W23U) still increases the total number of solved instances. This improvement is particularly notable in the weighted track, where WMaxCDCL-S6-HS12 already outperforms Eval and Cash by a significant margin, indicating that the remaining unsolved instances are especially hard to solve.

5.4.3 Empirical analysis

(W)MaxCDCL deals with two types of conflicts during its search: hard conflicts, which occur when a hard clause becomes empty, and soft conflicts, which arise when LB≥UB. Among the 2019 weighted instances, WMaxCDCL solves 118 instances during preprocessing without any conflict. Table 5.4 partitions the remaining 1901 instances according to the percentage of soft conflicts of an instance over the total number

of conflicts detected during the search of WMaxCDCL. We classify the instances in ranges of 20, e.g., column (80,60] contains instances with a percentage of soft conflicts between 80% (excluded) and 60% (included).

We identify the two most important groups of instances: the percentage of soft conflicts is at least 80% for 34% of instances, and is at most 20% for 39% of instances. Recall that WMaxCDCL+Unlock implements Alg. 12 in two stages. The first stage is executed without the unlocking mechanism, and the second stage is executed only if the first stage is not successful, i.e., if LB<UB. The success rate of the second stage is significantly different for these two groups of instances: 34% vs 9%, and 27 over the 33 additional instances solved using the unlocking mechanism (i.e., solved by WMaxCDCL+Unlock but not by WMaxCDCL) belong to the first group. The data for unweighted MaxSAT show a similar trend, but are omitted here due to the lack of space.

These data offer an intuitive explanation of the effectiveness of the unlocking mechanism: it is most beneficial when hard clauses, including those learned from previous soft conflicts, are not sufficiently effective in pruning the search space. In such cases, the unlocking mechanism should be used to cut subtrees as early as possible. The intricate relationship between hard clauses and the unlocking mechanism warrants further investigation.

In weighted MaxSAT, WMaxCDCL+Unlock outperforms WMaxCDCL by solving 4 or more additional instances in the following families: *scSequencing (7)*, *timetabling (4)* and *judgement(4)*. In unweighted MaxSAT, MaxCDCL+Unlock solves 5 or more additional instances compared to MaxCDCL in the following families: *optic(5)*, *mindset2 (5)* and *phylogeneric-trees(13)*.

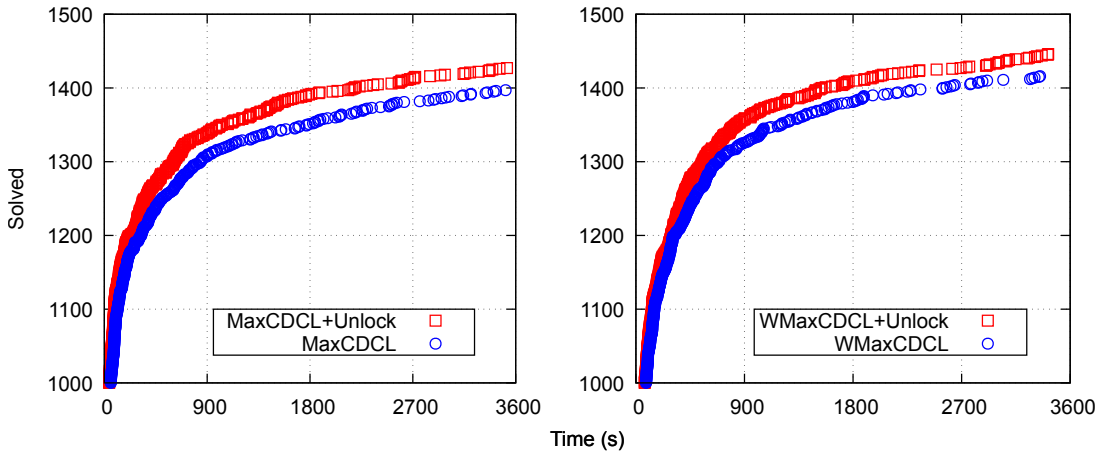


Figure 5.3: Number of instances solved with x seconds

5.5 Conclusions and Future Work

In this chapter, we proposed an unlocking mechanism that reuses soft clauses from existing cores to detect additional cores, thereby improving the lower bound for BnB MaxSAT solvers. The most challenging aspect of this mechanism is in weighted MaxSAT for calculating the weight of a new core d, the weight of a soft clause cl locked in d, and the remaining weight of cl for other cores, when d involves existing cores. We implemented the mechanism in the state-of-the-art solvers (W)MaxCDCL. Extensive experiments on all instances of the MaxSAT Evaluation from 2019 to 2023 demonstrate the effectiveness of the mechanism. We also provided an intuitive explanation on when and why the mechanism is effective. As future work, we plan to study the complex relations between the mechanism and hard clauses to further improve the effectiveness of the mechanism.

Algorithm 11: Lookahead(H, S, α, UB), the algorithm to compute LB

Input: H, a set of hard clauses; S, a set of unassigned soft literals; α, a partial assignment; UB, a given upper bound
Output: LB, a lower bound on the number of falsified soft literals for any extension of α.

LB ← number of soft literals falsified by α
$cores \leftarrow \emptyset$ `//will contain identified cores`
$H' \leftarrow H$ `//save the hard clauses`
while S is not empty and LB < UB **do**
  Pick s in S and propagate s
  for each falsified soft literal s' belonging to a core $c \in cores$ such that $w(c) > 0$ **do**
    $w(c) \leftarrow w(c) - 1$;
    if $w(c)$=0 **then**
      add all unassigned literals of c into S
    end if
  end for
  if an empty hard clause cl is produced or an unassigned soft literal s' in S is falsified **then**
    restore and save the weight of all cores in $cores$
    $d \leftarrow$ {propagated s starting a path to cl or s' in the implication graph G}
    add s' to d if cl is not produced
    $C \leftarrow \{c \in cores | \exists s \in d \; such \; that \; s \in c\}$
    $d \leftarrow d \cup \bigcup_{c \in C} c$
    $w(d) \leftarrow 1 + \sum_{c \in C} w(c)$
    LB++
    $cores \leftarrow (cores \cup \{d\}) \setminus C$
    $H \leftarrow H'$ `//restore the hard clauses`
    remove all locked literals from S
  end if
end while
return LB

Algorithm 12: WLookahead(H, (S, w), α, UB), the algorithm to compute LB for weighted MaxSAT

Input: H, a set of hard clauses; (S, w), a set of unassigned free soft literals with their weight function w; α, a partial assignment; UB, a given upper bound
Output: LB, a lower bound of total weight of falsified soft literals for any extension of α.
Notation: $c \prec s$ means c is unlocked before s is assigned or s is the last falsified soft literal fs

LB ← total weight of soft literals falsified by α
$cores \leftarrow \emptyset$; $H' \leftarrow H$ `//save the hard clauses`
$fs \leftarrow$ NULL; `//a falsified soft literal`
while S is not empty and LB < UB **do**
  Pick literal s in S and propagate s as assumption
  if a soft literal s' is falsified such that $rw(s') > 0$ or s' is in a core c with $wlock(s', c) > w(c)$ **then**
    $fs \leftarrow s'$
  else
    for each falsified soft literal s' in a core c such that $rw(s')=0$ and $w(c) > 0$ and $wlock(s', c) \leq w(c)$ **do**
      $w(c) \leftarrow w(c) - wlock(s', c)$;
      if $w(c)=0$ **then**
        add $\{u \mid u \in c$ and u is unassigned and $rw(c)=0\}$ into S
      end if
    end for
  end if
  if an empty hard clause cl is produced or fs is not NULL **then**
    $d \leftarrow$ {the propagated assumptions starting a path to cl or fs in the implication graph G}
    add fs into d if cl is not produced
    $C \leftarrow \{c \in cores \mid wlock(fs, c) > w(c)$ *or* c *unlocks* $s \in d\}$
    for each s in d **do**
      $aw(s) \leftarrow rw(s) + \sum_{c \in C, c \prec s}(wlock(s, c) - w(c))$
    end for
    $minaw \leftarrow min_{s \in d} aw(s)$
    for each s in d **do**
      $wlock(s, d) \leftarrow max(\sum_{c \in C} wlock(s, c), minaw)$
      $rw(s) \leftarrow max(0, aw(s) - wlock(s, d))$
    end for
    $w(c) \leftarrow ow(c)$ for each core c in $cores$
    $ow(d) \leftarrow w(d) \leftarrow minaw + \sum_{c \in C} ow(c)$
    $d \leftarrow d \cup \bigcup_{c \in C} c$
    LB ← LB $+ minaw$
    $cores \leftarrow (cores \cup \{d\}) \setminus C$
    remove all locked literals from S
    restore hard clauses and set fs to NULL
  end if
end while
return LB

Set	19∩20	19	20	21	22	23	Sum
M	102	322	290	323	183	177	1397
MU	102	331	298	330	185	181	1427
WM	95	281	315	378	203	144	1416
WMU	97	285	326	384	205	149	1446

Table 5.1: The number of instances solved by (W)MaxCDCL with and without the unlocking mechanism.

Set	19∩20	19	20	21	22	23	Sum
M23	116	350	327	360	190	180	1523
M23U	118	352	329	362	193	182	1536
Eval	118	350	330	361	191	189	1539
Cash	115	344	333	359	193	183	1527

Table 5.2: Results for benchmarks of the unweighted track.

Set	19∩20	19	20	21	22	23	Sum
W23	111	321	355	408	212	165	1572
W23U	111	318	355	409	219	166	1578
Eval	104	309	335	398	215	168	1529
Cash	110	311	349	402	213	154	1539

Table 5.3: Results for benchmarks of the weighted track.

soft confl (%)	[100,80]	(80,60]	(60,40]	(40,20]	(20,0]
#inst	645	185	142	187	742
#inst/sum	0.34	0.10	0.07	0.10	0.39
stage2 succs(%)	34	20	22	21	9
#unlock solved+	27	2	1	1	2

Table 5.4: Percentage range of soft conflicts (soft confl), number of instances for each range (#inst), percentage of instances for each range (#inst/sum), success rate for stage 2 lookahead (stage2 succ), additional instances solved thanks to unlocking mechanism for each range (#unlock solved+).

6 Tableau Calculus for MaxSAT

Sommaire

6.1 Preliminaries . . . 98
6.1.1 Boolean logic . . . 98
6.1.2 Signed logic . . . 99
6.1.3 Tableau calculi for SAT . . . 100
6.2 Related Work . . . 101
6.2.1 Tableau calculi for clausal MaxSAT . . . 101
6.2.2 Tableau calculi for non-clausal MaxSAT . . . 103
6.3 A Tableau Calculus for MaxSAT based on Resolution . . . 104
6.3.1 Soundness and Completeness . . . 107
6.3.2 Extending to the Weighted Partial MaxSAT . . . 109
6.4 A Tableau Calculus for Signed MaxSAT . . . 111
6.4.1 Soundness and Completeness . . . 112
6.4.2 Extending to the Weighted Partial Signed MaxSAT . . . 114
6.5 Conclusion . . . 114

The inference rules for SAT usually cannot be used in MaxSAT, as they preserve satisfiability but cannot preserve the minimum number of falsified clauses. For example, unit propagation, which is a successful SAT solving techniques, is unsound in MaxSAT, as illustrated in Example 6.1.

Example 6.1 *For a MaxSAT formula* $\phi = \{(x_1), (x_2), (x_3), (\neg x_1, \neg x_2), (\neg x_1, \neg x_3)\}$. *The optimal solution of* ϕ *is* $(0,1,1)$ *with only one falsified clause* (x_1). *But by propagating* x_1, *both* $\neg x_2$ *and* $\neg x_3$ *are propagated and there are two falsified clauses in the formula. The result shows that unit propagation is unsound in MaxSAT.*

Similarly, the resolution rule in SAT is also unsound in MaxSAT. Some relevant studies analyze how a refinement of this rule, can be applied in MaxSAT solving [30, 77, 95, 107, 2]. The extended resolution rule, or MaxSAT resolution rule, replaces two candidated clauses $(x, a_1, ..., a_n)$ and $(\neg x, b_1, ..., b_m)$ by their resolvent $(a_1, ..., a_n, b_1, ..., b_m)$ and $n+m$ compensation clauses, as shown in the follow. Compensation clauses preserves the number of falsified clauses, and it's proved complete: given a MaxSAT formula whose minimum number of falsified clauses is m, MaxSAT resolution will definitely derive a formula formed by exactly m empty clauses.

$$\frac{(x, a_1, ..., a_n) \qquad (\neg x, b_1, ..., b_m)}{\begin{array}{c}(a_1, ..., a_n, b_1, ..., b_m)\\ (x, a_1, ..., a_n, \neg b_1)\\ (x, a_1, ..., a_n, b_1, \neg b_2)\\ \dots\\ (x, a_1, ..., a_n, b_1, \dots, b_{m-1}, \neg b_m)\\ (\neg x, b_1, ..., b_m, \neg a_1)\\ (\neg x, b_1, ..., b_m, a_1, \neg a_2)\\ \dots\\ (\neg x, b_1, ..., b_m, a_1, \dots, a_{n-1}, \neg a_n)\end{array}}$$

MaxSAT resolution can produce a large number of derived clauses, which may expand the formula size significantly. This leads to inefficiency in solving large instances, as more clauses need to be processed, and it becomes computationally expensive. *The MaxSAT tableau calculus* is an alternative approach for solving MaxSAT. It is a inference system used to solve MaxSAT problems by systematically constructing a *tableau*, whose definition is listed below. Moreover, it is simpler and more intuitive than MaxSAT resolution in the sense that no compensation clauses are needed, and all the derived clauses are either empty or unit. Besides, as it can be extended to handle non-CNF formulas or more complex logical structures. The tableau can represent a wider variety of constraints and conditions, making it more expressive in certain scenarios.

Definition 6.1 *A tableau is a tree with a finite number of branches whose nodes are labelled by either a clause or a box (□). A box in a tableau denotes a contradiction. A branch is a maximal path in a tree, and we assume that branches have a finite number of nodes.*

This chapter is structured as follows. Section 6.1 defines the basic concepts. Section 6.2 reviews the related work, which includes the tableau calculi for SAT [59, 167], the calculi for MaxSAT [108] and for non-clausal MaxSAT [109]. In Section 6.3 we present our work, a novel MaxSAT tableau calculus based on resolution and proves its soundness and completeness. And we extend it to weighted partial maxsat in Section **??**. In Section 6.4 we extend the MaxSAT calculus to Signed MaxSAT and genelize it to weighted partial signed MaxSAT in Section **??**. We summarize our work in Section 6.5.

6.1 Preliminaries

6.1.1 Boolean logic

A propositional formula is an expression constructed from propositional variables using the propositional connectives $\wedge, \vee, \rightarrow$ and $\neg$ in accordance with the following rules: i) each propositional variable is a propositional formula; and ii) if A and B are propositional formulas, then so are $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, and $\neg A$.

A truth assignment is a mapping that assigns 0 (false) or 1 (true) to each propositional variable. A propositional formula is satisfied by an assignment if it is true under the usual truth-functional interpretation of the connectives and the truth values assigned to the variables.

Given a multiset of propositional formulas ϕ, *non-clausal SAT* is the problem of finding an assignment that satisfies all formulas, and *non-clausal MaxSAT* is the problem of finding an assignment of ϕ that minimizes the number of unsatisfied formulas. We use multisets instead of sets because duplicated formulas cannot be collapsed into a single formula; otherwise, the minimum number of unsatisfied formulas might not be preserved.

Clauses are a particular type of formulas: A clause is a disjunction of literals, where a literal l_i is a variable x_i or its negation $\neg x_i$. Non-clausal SAT is called *clausal SAT* when all the formulas in the multiset are clauses, and the same applies to MaxSAT.

A weighted formula is a pair (A, w), where A is a propositional formula and w, its weight, is a positive number. If A is a clause, (A, w) is a weighted clause. Given a multiset of weighted formulas (clauses) ϕ, *non-clausal (clausal) Weighted MaxSAT* is the problem of finding an assignment of ϕ that minimizes the sum of weights of unsatisfied formulas (clauses).

We can distinguish between hard and soft formulas. Soft formulas can be considered as formulas with a finite weight, and hard formulas can be considered as formulas with infinite weight, and are usually represented with weight $\top$. Given a multiset of hard and soft formulas (clauses) ϕ, *non-clausal (clausal) Weighted Partial MaxSAT* is the problem of finding an assignment of ϕ that satisfies all the hard formulas (clauses) and minimizes the sum of weights of unsatisfied soft formulas (clauses).

6.1.2 Signed logic

The underlying basic idea of Signed logic is to generalize the classical notion of literal by adorning an atomic formula with a sign which consists of a finite set of truth values.

Boolean logic can be generalized to the many-valued setting by defining a more general notion of literal, called signed literal. Given a finite truth value set $N_n = \{0, \frac{1}{n-1}, ..., \frac{n-1}{n-1}\}$, a *sign* S is a subset of N_n, and a *signed literal* is an expression of the form $S{:}x$, where S is a sign and x is a propositional variable. The complement of a signed literal $l = S{:}x$ is $\neg l = \neg S{:}x = (N_n \setminus S){:}x$.

A *signed formula* is an expression constructed from signed literals using the propositional connectives $\wedge, \vee, \rightarrow$ and $\neg$ in accordance with the following rules: i) each signed literal is a signed formula; and ii) if A and B are signed formulas, then so are $\neg A$, $(A \vee B)$, $(A \wedge B)$, and $(A \rightarrow B)$. For the particular case of signed formulas in Conjunctive Normal Form (CNF), we have that a *signed clause* is a disjunction of signed literals, and a *signed CNF formula* is a multiset of signed clauses.

A truth assignment for a signed formula is a mapping that assigns to each propositional variable an element of the truth value set. An assignment I satisfies a signed literal $S{:}x$ iff $I(x) \in S$, and satisfies a signed formula if it is true under the truth values assigned to the variables and the following interpretation of the con-

Table 6.1: α-formulas and β-formulas.

α	α_1	α_2
$A \wedge B$	A	B
$\neg(A \vee B)$	$\neg A$	$\neg B$
$\neg(A \rightarrow B)$	A	$\neg B$

β	β_1	β_2
$A \vee B$	A	B
$\neg(A \wedge B)$	$\neg A$	$\neg B$
$A \rightarrow B$	$\neg A$	B

nectives: $I(\neg A) = 1 - I(A)$, $I(A \vee B) = \max\{I(A), I(B)\}$, $I(A \wedge B) = \min\{I(A), I(B)\}$, and $I(A \rightarrow B) = \max\{1 - I(A), I(B)\}$. In the case of signed CNF formula, it satisfies a signed clause c iff it satisfies at least one of the signed literals in c, and satisfies a signed CNF formula ϕ iff it satisfies all clauses in ϕ. The Signed SAT problem for a set of signed clauses ϕ is to find an assignment that satisfies all the signed clauses in ϕ. See example 6.2 for an instance.

Example 6.2 *Given the truth value set* $N_3 = \{0, \frac{1}{2}, 1\}$*, it's clear that the literals* $\{0, \frac{1}{2}\}$*:*x *and* $\{1\}$*:*x *are complementary literals. The signed formula* $(\{0, \frac{1}{2}\}:x \vee \{0, \frac{1}{2}\}:y) \wedge (\{0,1\}:x \vee \{0,1\}:y) \wedge (\{\frac{1}{2}, 1\}:x \vee \{\frac{1}{2}, 1\}:y)$ *encodes that* x *and* y *cannot take the same value.*

Given a multiset of signed formulas ϕ, non-clausal Signed MaxSAT is to find an assignment that minimizes the number of unsatisfied signed formulas in ϕ. If ϕ is composed of hard formulas and weighted soft formulas, non-clausal Weighted Partial Signed MaxSAT is the problem of finding an assignment of ϕ that satisfies all the hard formulas and minimizes the sum of weights of unsatisfied soft formulas. When the formulas are clauses, we talk about clausal Weighted Partial Signed MaxSAT.

Remark: We define Signed logic so that atomic formulas are signed literals. We could also define it so that we associate a sign with each propositional formula. We use the first option because it is the most natural non-clausal extension of signed CNF formulas [14].

6.1.3 Tableau calculi for SAT

In the uniform notation, all propositional formulas of the form $(A \circ B)$ and $\neg(A \circ B)$, where A and B denote propositional formulas and $\circ \in \{\vee, \wedge, \rightarrow\}$, are grouped into two categories so that the definition of inference rules and the proofs are simplified: Those that act conjunctively, which are called α-formulas, and those that act disjunctively, which are called β-formulas. The different formulas in each category are displayed in Table 6.1. To complete a taxonomy of propositional formulas, excluding literals, we also need the propositional formulas of the form $\neg(\neg A)$.

Note that α is logically equivalent to $\alpha_1 \wedge \alpha_2$, β is logically equivalent to $\beta_1 \vee \beta_2$ and $\neg(\neg A)$ is logically equivalent to A. In SAT tableaux, these equivalences are used to reduce the problem of finding a satisfying assignment of α to that of finding a

Table 6.2: Tableau expansion rules for SAT

$\dfrac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}}$	$\dfrac{\beta}{\beta_1 \mid \beta_2}$	$\dfrac{\neg(\neg A)}{A}$	$\dfrac{l \quad \neg l}{\square}$
α-rule	β-rule	$\neg$-rule	$\square$-rule

Table 6.3: n-ary β-rule

$$\dfrac{l_1 \vee l_2 \vee \cdots \vee l_n}{l_1 \mid l_2 \mid \cdots \mid l_n}$$

n-ary β-rule

satisfying assignment of both α_1 and α_2, of β to that of finding a satisfying assignment of β_1 or β_2, and of $\neg(\neg A)$ to that of finding a satisfying assignment of A. Thus, using the expansion rules of Table 6.2 we obtain a complete tableau calculus for non-clausal SAT. We introduced the *contradiction rule* ($\square$-rule), where l denotes a literal, because it will be necessary in MaxSAT; in the literature, applying this rule is usually referred to as closing the branch.

The tableau method is used to determine the satisfiability of a given set of propositional formulas [167]. It starts creating an initial tableau composed of a single branch that has a node for each formula in the input set of formulas. Then, it applies the expansion rules of Table 6.2 until a contradiction is derived in each branch (in this case, the input set of formulas is unsatisfiable) or a branch is saturated without deriving a contradiction (in this case, the input set of formulas is satisfiable). A branch is *saturated* in a SAT tableau when all the possible applications of the expansion rules have been applied in that branch.

6.2 Related Work

6.2.1 Tableau calculi for clausal MaxSAT

The first tableau MaxSAT calculus in the literature [108] is for clausal MaxSAT. Thus, it does not contain the α and $\neg$-rules in Table 6.2. It consists of the β- and $\square$-rule. In fact, as all the formulas in the tableau are clauses and the formulas of type β are always disjunctions of literals of the form $l_1 \vee l_2 \vee \cdots \vee l_n$, the previous β-rule is replaced with the n-ary β-rule of Table 6.3. Note that the n-ary β-rule collapses $n-1$ applications of the β-rule over the clause $l_1 \vee l_2 \vee \cdots \vee l_n$.

The expansion rules are applied differently in MaxSAT: Firstly, the application of expansion rules in a branch cannot stop once a contradiction is detected. Since the

aim of MaxSAT is to derive all the possible contradictions, the application of rules in a branch must continue until no more expansion rules can be applied. Thus, we will say that a tableau is completed when all the branches are saturated. Secondly, the application of rules in SAT leads to accumulate the newly added unit clauses in the branch in such a way that satisfiability is preserved in at least one branch when the input set of clauses is satisfiable. However, the addition of unit clauses can lead to wrong MaxSAT solutions. In clause MaxSAT tableaux, the goal should be to keep the minimum number of unsatisfied clauses in at least one branch and not to decrease that number in the rest of branches. To this end, we must maintain *active* and *inactive* clauses in a branch. Once a clause has been used as a premise of a rule in a branch, it cannot be used again in that branch and becomes inactive. For example, thanks to distinguishing between active and inactive clauses, we will detect one contradiction in the multiset of unit clauses $\{x_1, \neg x_1, \neg x_1\}$ and two in $\{x_1, x_1, \neg x_1, \neg x_1\}$. Without that, we could detect two contradictions in the first case, obtaining a wrong answer. Indeed, the inference rules for MaxSAT can be seen as rewriting rules.

Figure 6.1 shows the differences between clausal SAT and clausal MaxSAT tableaux using the multiset $\phi = \{\neg x_1, \neg x_2, \neg x_3, x_1 \vee x_2, \neg x_1 \vee x_3\}$. In SAT, it is enough with applying the β-rule to $x_1 \vee x_2$. Since a contradiction is detected in each branch, the input multiset of formulas is declared unsatisfiable. However, in MaxSAT, the β-rule must also be applied to $\neg x_1 \vee x_3$ and all the possible contradictions must be detected to complete the tableau. Note that in the leftmost branch of the MaxSAT tableau there is just one contradiction because we have just one occurrence of x_1, which became inactive after detecting the first contradiction. Each literal can only be used once to detect a contradiction.

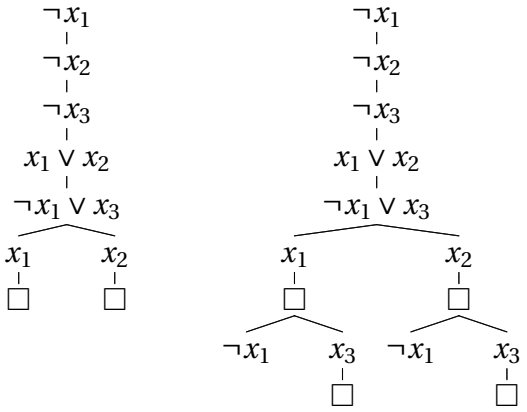


Figure 6.1: Completed clausal SAT tableau (left) and completed clausal MaxSAT tableau (right) for the multiset of clauses $\phi = \{\neg x_1, \neg x_2, \neg x_3, x_1 \vee x_2, \neg x_1 \vee x_3\}$.

The soundness of the previous clausal MaxSAT tableau calculus states that the β and $\square$-rule preserve the minimum number of unsatisfied clauses between a tableau and its extension; in particular, the β-rule preserves that number in at least one

Table 6.4: Tableau expansion rules for non-clausal MaxSAT

α	β	$\neg(\neg A)$	l
$\square \mid \alpha_1$ α_2	$\beta_1 \mid \beta_2$	A	$\neg l$ $\square$
α-rule	β-rule	$\neg$-rule	$\square$-rule

branch and does not decrease it in the rest of the branches. So, once all branches have been saturated, the minimum number of contradictions derived among the branches of a completed tableau is the minimum number of unsatisfied clauses in the input multiset of clauses. The completeness states that any completed tableau for a multiset of clauses ϕ, whose minimum number of clauses that can be unsatisfied in it is m, has a branch with m contradictions and the rest of branches contain at least m contradictions [108]. Thus, in Figure 6.1, the right tableau proves that the minimum number of unsatisfied clauses in ϕ is 1.

6.2.2 Tableau calculi for non-clausal MaxSAT

If we want to deal with arbitrary propositional formulas (i.e., non-clausal MaxSAT), the first drawback is that the α-rule does not preserve the minimum number of unsatisfied formulas as the β-rule does for clauses. Assume that we want to solve the non-clausal MaxSAT instance $\{x_1, x_2, \neg x_1 \wedge \neg x_2\}$, whose single optimal assignment is the one that sets x_1 and x_2 to true and only unsatisfies $\neg x_1 \wedge \neg x_2$. If we apply the α-rule to $\neg x_1 \wedge \neg x_2$, we add two nodes, labelled with $\neg x_1$ and $\neg x_2$, to the initial tableau. Then, we can derive two contradictions by applying the $\square$-rule to $\{x_1, \neg x_1\}$ and $\{x_2, \neg x_2\}$, but the minimum number of formulas unsatisfied by the optimal assignment is just one. This counterexample shows that the α-rule is unsound in MaxSAT. So, we need to define a new and sound α-rule as a first step towards getting a sound and complete non-clausal MaxSAT calculus.

The first defined complete non-clausal MaxSAT tableau calculus is formed by the expansion rules in Table 6.4 [109]. Note that all the rules preserve the number of premises unsatisfied by an assignment I in at least one branch and do not decrease that number in the other branch (if any). In particular, in the α-rule, we have that if I unsatisfies α, the left branch contains one contradiction and α_1 and α_2 cannot be used to derive any other contradiction in that branch because they are not expanded; moreover, I unsatisfies α_1 or α_2 (or both) on the right branch. On the other hand, if I satisfies α, then I also satisfies α_1 and α_2 on the right branch. A similar non-clausal MaxSAT tableau calculus was later proposed by Fiorino [68].

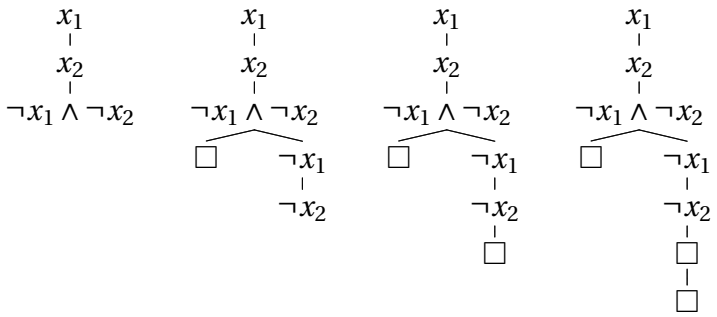


Figure 6.2: A non-clausal MaxSAT tableau for $\{x_1, x_2, \neg x_1 \wedge \neg x_2\}$.

Example 6.3 *Figure 6.2 shows how to construct a MaxSAT tableau for the multiset $\phi = \{x_1, x_2, \neg x_1 \wedge \neg x_2\}$. We first construct the initial tableau (the leftmost tableau) and then apply the α-rule to $\neg x_1 \wedge \neg x_2$, getting as a result the second tableau. The leftmost branch is saturated and we apply the $\square$-rule to $\{x_1, \neg x_1\}$ on the rightmost branch, getting as a result the third tableau. Finally, we apply the $\square$-rule to $\{x_2, \neg x_2\}$ on the same branch and get the rightmost tableau in the figure. Since the minimum number of contradictions among the branches of the last tableau is 1, the minimum number of formulas that can be unsatisfied in ϕ is 1.*

6.3 A Tableau Calculus for MaxSAT based on Resolution

The MaxSAT tableau calculus we introduce in section 6.2 are not always efficient, as illustrated by example 6.4. It takes time to prove that its minimum number of unsatisfied clauses is one, and the size of final tableau is big.

Example 6.4 *For the multiset of clauses $\{(x_1), (x_2), (\neg x_1 \vee x_3), (\neg x_1 \vee \neg x_2 \vee \neg x_3)\}$, we use the β-rule and $\square$-rule in clausal MaxSAT tableau to prove that the minimum number of unsatisfied clauses in it is one. Figure 6.3 shows the process. We first creat the initial tableau and the apply the β-rule on clause $(\neg x_1 \vee x_3)$, getting the left branch $\neg x_1$ and right branch x_3. We apply $\square$-rule to x_1 and $\neg x_1$ on the left branch and get a contradiction. Then, on the left branch, we apply β-rule to clause $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$ and get $\neg x_2$ and $\neg x_1 \vee \neg x_3$ as two branches. By applying $\square$-rule to x_2 and $\neg x_2$ we get a new contradiction on the leftmost branch, and this branch now is saturated. By applying β-rule on $\neg x_1 \vee \neg x_3$ we get two branches $\neg x_1$ and $\neg x_3$, both of them are saturated because no more rule can be applied to them. Then, we apply β-rule to clause $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$ on the right main branch, and get $\neg x_1$ and $\neg x_2 \vee \neg x_3$ as two branches. By applying $\square$-rule to x_1 and $\neg x_1$ we get a contradiction and this branch is saturated. For the rightmost branch $\neg x_2 \vee \neg x_3$, β-rule divides it to $\neg x_2$ and $\neg x_3$. Both deduce a contradiction by $\square$-rule and become saturated. Since the minimum number of $\square$ among all branches is one, the minimum number of falsified clauses is also one.*

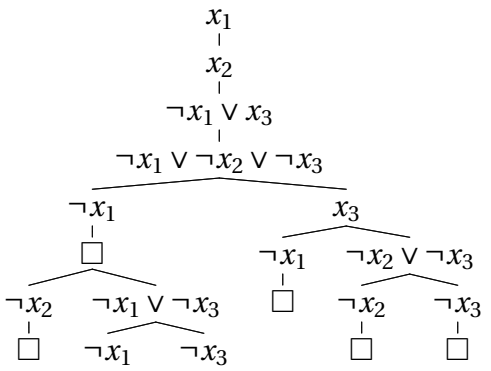


Figure 6.3: A tableaux for a MaxSAT instance $\{(x_1),(x_2),(\neg x_1 \vee x_3),(\neg x_1 \vee \neg x_2 \vee \neg x_3)\}$.

In this section, we define a new MaxSAT tableau calculus, which is based on resolution rule, and prove its soundness and completeness. The proposed calculus has the advantage of producing shorter proofs in some cases, as shown in example 6.5.

Definition 6.2 *Let $\phi = \{\phi_1, \ldots, \phi_m\}$ be a multiset of clauses, l a literal, and D and D' disjunction of literals. A tableau for ϕ is constructed by a sequence of applications of the following rules:*

Initialize *A tree with a single branch with m nodes such that each node is labelled with a clause of ϕ is a tableau for ϕ. Such a tableau is called the initial tableau and its clauses are declared to be active.*

Given a tableau T for ϕ and a branch b of T,

Res-rule *If b contains two active clauses with complementary literals, $l \vee D$ and $\neg l \vee D'$, the tableau obtained by appending a new left branch with two nodes below b labelled with $\neg l$ and D and a new right branch with two nodes below b labelled with l and D' is a tableau for ϕ. Clauses $l \vee D$ and $\neg l \vee D'$ become inactive in b and the added clauses are declared to be active.*

Unit-rule *If b contains an active unit clause l and an active non-unit clause $\neg l \vee D$, the tableau obtained by appending a new left node below b labelled with $\square$ and a new right node with two nodes below b labelled with l and D is a tableau for ϕ. Clauses l and $\neg l \vee D$ become inactive in b and the added non-empty clauses are declared to be active.*

$\square$-rule *If b contains two active and complementary unit clauses, l and $\neg l$, the tableau obtained by appending a node below b labelled with $\square$ is a tableau for ϕ. Clauses l and $\neg l$ become inactive in b.*

The expansion rules of the previous definition are summarized in Figure 6.4. Note that all the rules preserve the number of premises falsified by an assignment I in at least one branch and do not decrease that number in the other branch (if any).

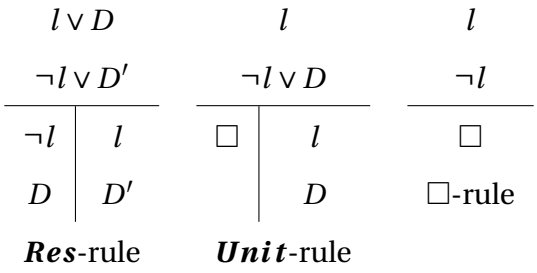


Figure 6.4: Tableau expansion rules for non-clausal MaxSAT

Definition 6.3 *Let T be a MaxSAT tableau for a multiset of propositional clauses ϕ. A branch b of T is saturated when no further expansion rules can be applied on b, and T is completed when all its branches are saturated. The cost of a saturated branch is the number of boxes on the branch. The cost of a completed tableau is the minimum cost among all its branches.*

Notice that a branch becomes saturated when it does not contain active clauses with complementary literals. We show below that the minimum number of clauses that can be unsatisfied in a multiset of propositional clauses ϕ is m iff the cost of a completed tableau for ϕ is m. Thus, the systematic construction of a completed tableau for ϕ provides an exact method for MaxSAT.

Example 6.5 *Figure 6.5 shows how we can create a tableau, with the previous calculus, to prove that the minimum number of unsatisfied clauses in the multiset of clauses $\{x_1, x_2, \neg x_1 \vee x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3\}$ is one. We first create the initial tableau (the leftmost tableau) and then apply the **Res**-rule to the clauses $\neg x_1 \vee x_3$ and $\neg x_1 \vee \neg x_2 \vee \neg x_3$ (resolving variable x_3), getting as a result the second tableau in the figure. We apply the $\square$-rule to x_1 and $\neg x_1$ on the leftmost branch and obtain the third tableau. That branch is now saturated because its current active clauses (x_2 and $\neg x_3$) do not contain complementary literals. Then, we apply the **Unit**-rule to the clauses x_1 and $\neg x_1 \vee \neg x_2$ (resolving variable x_1) on the rightmost branch, getting as a result the fourth tableau whose middle branch is saturated (current active clauses: x_2 and x_3). Finally, we apply the $\square$-rule to x_2 and $\neg x_2$ on the rightmost branch and this branch becomes also saturated (current active clauses: x_1 and x_3). Since the minimum number of boxes among the branches of the last tableau is one, the minimum number of clauses that can be unsatisfied in ϕ is also one.*

The advantage of the defined calculus with respect to other MaxSAT tableau calculi [68, 108, 109] is that it does not need to expand all the active clauses to saturate a branch. It only needs to expand those active clauses containing a complementary literal in another active clause of the branch. This implies, in some cases, that the resulting tableaux have fewer nodes. For instance, the other calculi need to double the number of branches to solve the multiset of clauses of Example 6.5.

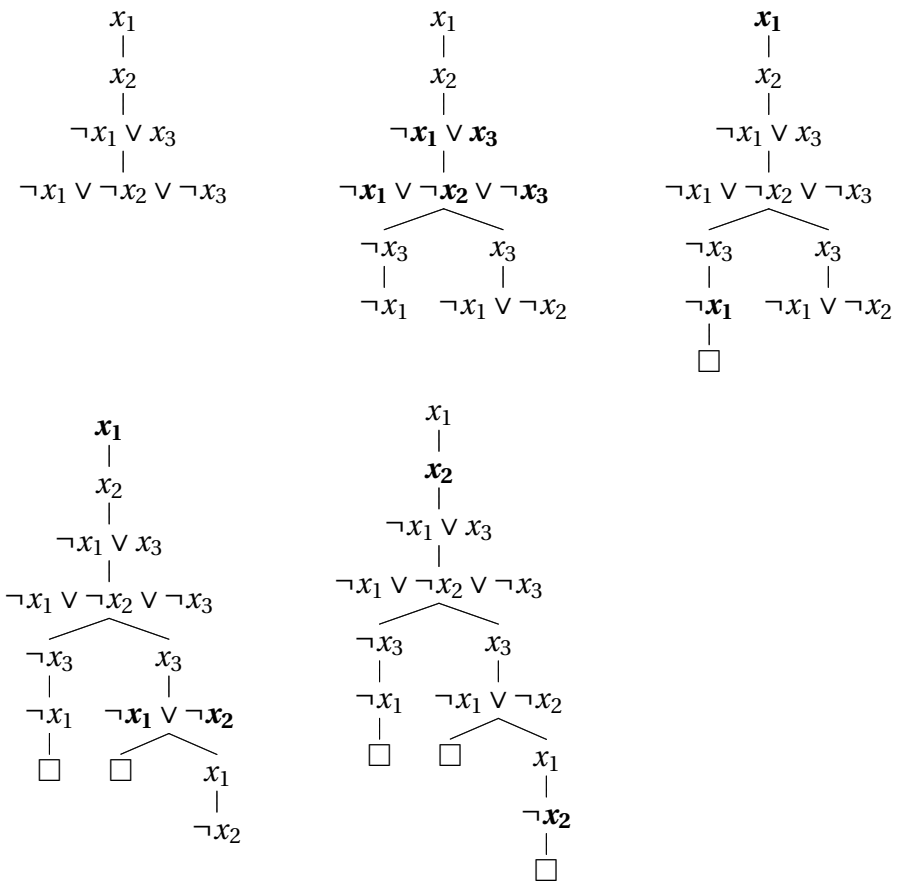


Figure 6.5: A tableaux for the non-clausal MaxSAT instance $\{x_1, x_2, \neg x_1 \vee x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3\}$.

6.3.1 Soundness and Completeness

We prove the soundness and completeness of the proposed tableau calculus for MaxSAT. Before presenting the completeness theorem, we prove termination and the soundness of the expansion rules.

Proposition 6.1 *A tableau for a multiset of propositional clauses ϕ is completed in a finite number of steps.*

Proof 6.1 *We first create an initial tableau and then apply expansion rules in the newly created branches until they become saturated. The **Res**- and **Unit**-rule reduce the number of connectives. Since we began with a finite number of connectives, these rules can only be applied a finite number of times. The $\square$-rule inactivates two literals and adds a box. Since we began with a finite number of literals and boxes cannot be premises of any expansion rule, this rule can only be applied a finite number of times. Hence, the construction of any completed tableau terminates in a finite number of steps.*

Proposition 6.2 *An assignment I falsifies k premises of a* ***Res****-,* ***Unit****-, and* $\square$*-rule iff assignment I falsifies k clauses in one branch of the conclusions of the rule and at least k clauses in the other branch (if any).*

Proof 6.2 *We prove the result for each rule:*

- ***Res****-rule: An assignment I satisfies both* $l \vee D$ *and* $\neg l \vee D'$ *iff I satisfies either* l *and* D' *or* $\neg l$ *and* D*. In this case, I satisfies the clauses of one branch and falsifies at least one clause of the other branch. In any other case, I falsifies either* $l \vee D$ *or* $\neg l \vee D'$*. If I falsifies* $l \vee D$*, it falsifies exactly one clause (*D*) of the left branch and at least one clause of the right branch. If I falsifies* $\neg l \vee D'$*, it falsifies exactly one clause (*D'*) of the right branch and at least one clause of the left branch. Hence, the number of unsatisfied clauses is preserved in at least one branch of the rule.*

- ***Unit****-rule: An assignment I satisfies both* l *and* $\neg l \vee D$ *iff I satisfies* l *and* D*, and in this case I satisfies the two clauses of the right branch. In any other case, I falsifies either* l *or* $\neg l \vee D$*. So, I falsifies the left branch and at least one clause of the right branch. Hence, the number of unsatisfied clauses is preserved in at least one branch of the rule.*

- $\square$*-rule: An assignment I either falsifies* l *or* $\neg l$*, and satisfies the complementary literal of the unsatisfied literal. Since the single conclusion is a box and denotes a contradiction, I falsifies the same number of clauses in the premises and the conclusion.*

Theorem 6.1 ***Soundness & completeness.*** The cost of a completed tableau for a multiset of clauses ϕ is m iff the minimum number of unsatisfied clauses in ϕ is m.

Proof 6.3 *(*Soundness:*) T was derived by creating a sequence of tableaux $T_0, \ldots, T_n$ ($n \geq 0$) such that T_0 is an initial tableau for ϕ, $T_n = T$, and T_i was obtained by a single application of the* ***Res****-,* ***Unit****- or* $\square$*-rule on an branch of T_{i-1} for $i = 1, \ldots, n$. By Proposition 6.1, we know that such a sequence is finite. Since T has cost m, T_n contains one branch b with exactly m boxes and the rest of branches contain at least m boxes. Moreover, the active clauses in every branch of T_n do not contain complementary literals; otherwise, we could yet apply expansion rules and T_n could not be completed. The assignment that sets to true the literals occurring in the active clauses of an optimal branch only falsifies the m boxes and there cannot be any assignment satisfying less than m clauses in a branch of T_n because each branch contains at least m boxes. Therefore, the minimum number of active clauses than can be unsatisfied among the branches of T_n is m.*

Proposition 6.2 guarantees that the minimum number of unsatisfied active clauses is preserved in the sequence of tableaux $T_0, \ldots, T_n$. Thus, the minimum number of unsatisfied active clauses in T_0 is also m. Since T_0 is formed by a single branch that only contains the clauses in ϕ and all these clauses are active, the minimum number of clauses that can be unsatisfied in ϕ is m.

*(*Completeness:*) Assume that there is a completed tableau T for ϕ that does not have cost m. We distinguish two cases:*

(i) T has a branch b of cost k, where $k < m$. Then, T has a branch with k boxes and a satisfiable multiset of active clauses because T is completed. This implies that the minimum number of unsatisfied active clauses among the branches of T is at most k. By Proposition 6.2, this also holds for T_0, but this is in contradiction with m being the minimum number of unsatisfied clauses in ϕ because $k < m$. Thus, any branch of T has at least cost m.

(ii) T has no branch of cost m. This is in contradiction with m being the minimum number of unsatisfied clauses in ϕ. Since the tableau expansion rules preserve the minimum number of unsatisfied clauses and the branches of any completed tableau only contain active clauses that are boxes or clauses without complementary literals, T must have a saturated branch with m boxes. Thus, T has a branch of cost m.

Hence, each completed tableau T for a multiset of clauses ϕ has cost m if the minimum number of clauses that can be unsatisfied in ϕ is m.

6.3.2 Extending to the Weighted Partial MaxSAT

Dealing with weighted soft clauses can be understood as collapsing several unweighted MaxSAT inferences into a single inference, because a weighted clause (C, w) can be replaced by w copies of the unweighted clause C. If there are two premises (C_1, w_1) and (C_2, w_2) with different weights $(w_1 \neq w_2)$, (C_1, w_1) and (C_2, w_2) become inactive but $(C_1, w_1 - w)$ and $(C_2, w_2 - w)$, where $w = \min(w_1, w_2)$, are added as active clauses (clauses with weight 0 are not added). Then, the conclusions of the inference have weight w. For example, from $(x_1, 1)$ and $(\neg x_1, 3)$ we derive $(\square, 1)$ and $(\neg x_1, 2)$.

When dealing with hard clauses, the inference applied in SAT remains valid in MaxSAT when the premises are hard. Moreover, the detection of a contradiction between two hard clauses implies that we have identified an infeasible solution. In this case, the contradiction is represented by $\blacksquare$ and the branch containing that contradiction can be pruned.

Figure 6.6 displays the expansion rules of a complete tableau calculus for Weighted Partial MaxSAT. It is formed by the extensions of the ***Res***-, ***Unit***-, and $\square$-rules when their premises contain unit hard clauses or weighted soft clauses. In the case in which we have a non-unit hard clause, we can use the β-rule. In our calculus, unit hard clauses are always active while non-unit hard premises become inactive after applying an inference rule. Notice that other inference rules could be used to deal with hard premises but we used the β-rule because it produces a simple and complete calculus.

Example 6.6 *Figure 6.7 displays a tableau for the WPMaxSAT instance* $\{(\neg x_1 \vee \neg x_2, \top), (\neg x_2 \vee \neg x_3, 2), (x_1, 2), (x_2, 3), (x_3, 2)\}$. *Firstly, we apply the β-rule to* $(\neg x_1 \vee \neg x_2, \top)$. *Secondly, we apply the $\square$-rule to* $(\neg x_1, \top)$ *and* $(x_1, 2)$ *in the left branch. Thirdly, we apply the $\square$-rule to* $(\neg x_2, \top)$ *and* $(x_2, 3)$ *in the right branch. Fourthly, we apply the $Unit$-rule to* $(\neg x_2 \vee \neg x_3, 2)$ *and* $(x_2, 3)$ *in the left branch. Fifthly, we apply the $\square$-rule to* $(\neg x_3, 2)$ *and* $(x_3, 2)$ *in the second leftmost branch. Sixthly, we apply the $Unit$-rule*

$$\frac{(l \vee D, \top)}{(l, \top) \mid (D, \top)}$$

$\boldsymbol{\beta}$-rule

$$\frac{\begin{array}{c}(l \vee D, w_1)\\ (\neg l \vee D', w_2)\end{array}}{\begin{array}{c|c}(l \vee D, w_1 - w) & (l \vee D, w_1 - w)\\ (\neg l \vee D', w_2 - w) & (\neg l \vee D', w_2 - w)\\ (\neg l, w) & (l, w)\\ (D, w) & (D', w)\end{array}}$$

where $w = \min(w_1, w_2)$

$\boldsymbol{Res}$-rule

$$\frac{\begin{array}{c}(l, \top)\\ (\neg l \vee D, w)\end{array}}{(D, w)}$$

$$\frac{\begin{array}{c}(l, w_1)\\ (\neg l \vee D, w_2)\end{array}}{\begin{array}{c|c}(l, w_1 - w) & (l, w_1 - w)\\ (\neg l \vee D, w_2 - w) & (\neg l \vee D, w_2 - w)\\ (\square, w) & (l, w)\\ & (D, w)\end{array}}$$

where $w = \min(w_1, w_2)$

$\boldsymbol{Unit}$-rule

$$\frac{\begin{array}{c}(l, \top)\\ (\neg l, \top)\end{array}}{\blacksquare} \qquad \frac{\begin{array}{c}(l, \top)\\ (\neg l, w)\end{array}}{(\square, w)} \qquad \frac{\begin{array}{c}(l, w_1)\\ (\neg l, w_2)\end{array}}{\begin{array}{c}(\square, w)\\ (l, w_1 - w)\\ (\neg l, w_2 - w)\end{array}}$$

where $w = \min(w_1, w_2)$

$\square$-rule

Figure 6.6: Tableau expansion rules for Weighted Partial MaxSAT

to $(\neg x_2 \vee \neg x_3, 2)$ *and* $(x_3, 2)$ *in the right branch. Since the minimum cost among all the branches is 3, the minimum sum of weights of the unsatisfied soft clauses while satisfying the hard clauses is 3.*

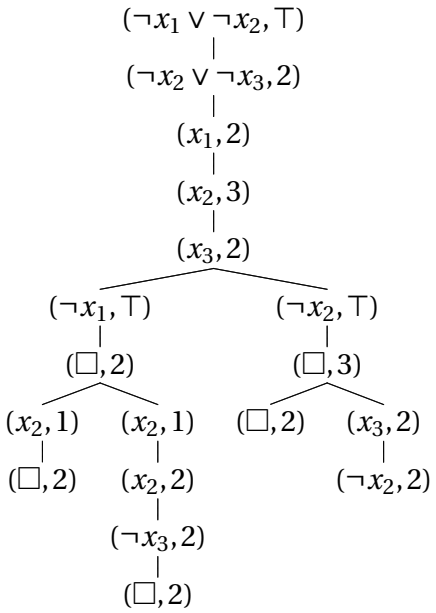


Figure 6.7: A tableau for the WPMaxSAT instance $\{(\neg x_1 \vee \neg x_2, \top), (\neg x_2 \vee \neg x_3, 2), (x_1, 2), (x_2, 3), (x_3, 2)\}$

Taking into account that a soft clause (c, w) is equivalent to having w copies of clause $(c, 1)$, and $\{(c, w_1), (c, w_2)\}$ is equivalent to $(c, w_1 + w_2)$, we can prove that the previous calculus is complete for WPMaxSAT.

6.4 A Tableau Calculus for Signed MaxSAT

To extend the boolean MaxSAT tableau calculi to signed MaxSAT, we must bear in mind that signed logic is like Boolean logic above the literal level. Thus, the α, β and $\neg$-rules of Table 6.4 remain sound for Signed MaxSAT but the $\square$-rule must be redefined. The first observation is that we may need more than two literals to detect unsatisfiability. For example, assuming $N_3 = \{0, \frac{1}{2}, 1\}$, the unsatisfiable multiset $\{\{0, \frac{1}{2}\}{:}x, \{0, 1\}{:}x, \{\frac{1}{2}, 1\}{:}x\}$, composed of three signed literals, becomes satisfiable if any of its literals is removed. The second observation is that we must add some compensation literals when we detect a contradiction in order to preserve the minimum number of unsatisfied formulas. For example, given the multiset $\{\{0\}{:}x, \{\frac{1}{2}\}{:}x, \{1\}{:}x\}$ and assuming that the truth value set is N_3, if we derive a contradiction from the first two literals, we obtain $\{\square, \{1\}{:}x\}$, whose minimum number of unsatisfied literals is 1. However, it is easy to realize that the minimum number of unsatisfied literals in $\{\{0\}{:}x, \{\frac{1}{2}\}{:}x, \{1\}{:}x\}$ is 2 because any interpretation assigns to x a single value of N_3.

Table 6.5 contains the inference rules of our sound and complete tableau calculus for Signed MaxSAT. Notice that the $\square$-rule has the proviso that the intersection of the signs of the signed literals in the premise must be empty. Although it is not necessary for completeness, we could also impose that the premises of the $\square$-rule are a minimum

Table 6.5: Tableau expansion rules for Signed MaxSAT

$$\frac{\alpha}{\begin{array}{c|c}\square & \alpha_1 \\ & \alpha_2\end{array}}\ \alpha\text{-rule} \qquad \frac{\beta}{\begin{array}{c|c}\beta_1 & \beta_2\end{array}}\ \beta\text{-rule} \qquad \frac{\neg(\neg A)}{A}\ \neg\text{-rule}$$

$$\frac{\begin{array}{c} S_1{:}x \\ S_2{:}x \\ \dots \\ S_m{:}x \end{array}}{\square}$$

$S_1 \cup S_2{:}x$

$(S_1 \cap S_2) \cup S_3{:}x$

$\dots\dots\dots$

$(S_1 \cap \dots \cap S_{m-1}) \cup S_m{:}x$

provided that $S_1 \cap \dots \cap S_m = \emptyset$

$\square$-rule

unsatisfiable set of signed literals. Moreover, the conclusion of the rule contains, apart from the contradiction, a number of compensation signed literals to ensure the preservation of the minimum number of unsatisfied formulas.

As in Boolean MaxSAT, a branch is saturated when no more expansion rules can be applied on it and a tableau is completed when all its branches are saturated. The soundness and completeness state that the minimum number of unsatisfied formulas in a multiset of signed formulas ϕ is m iff the minimum number of contradictions among the branches of any completed tableau of ϕ (its cost) is m.

Example 6.7 *Given N_3, the application of the $\square$-rule to $\{\{0\}{:}x, \{\frac{1}{2}\}{:}x, \{1\}{:}x\}$ derives $\{\square, \{0, \frac{1}{2}\}{:}x,\ \{1\}{:}x\}$, and the application of the $\square$-rule to the derived multiset derives $\{\square, \square, \{0, \frac{1}{2}, 1\}{:}x\}$=$\{\square, \square\}$ ($\{0, \frac{1}{2}, 1\}{:}x$ is a tautology).*

Given the multiset $\{\{0, \frac{1}{2}\}{:}x, \{0, 1\}{:}x, \{\frac{1}{2}, 1\}{:}x\}$, the $\square$-rule derives $\{\square, \{0, \frac{1}{2}, 1\}{:}x, \{0, \frac{1}{2}, 1\}{:}x\} = \{\square\}$.

Example 6.8 *Figure **??** shows the construction of a completed Signed MaxSAT tableau for $\{\{\frac{1}{2}\}{:}x \wedge \{\frac{1}{2}\}{:}y, \{0, 1\}{:}x \vee \{0, 1\}{:}y\}$. Firstly, the α-rule is applied to $\{\frac{1}{2}\}{:}x \wedge \{\frac{1}{2}\}{:}y$. Secondly, the β-rule is applied to $\{0, 1\}{:}x \vee \{0, 1\}{:}y$ on the leftmost branch. Thirdly, the β-rule is applied to $\{0, 1\}{:}x \vee \{0, 1\}{:}y$ on the rightmost branch. Fourthly, the $\square$-rule is applied to $\{\frac{1}{2}\}{:}x$ and $\{0, 1\}{:}x$. Fifthly, the $\square$-rule is applied to $\{\frac{1}{2}\}{:}y$ and $\{0, 1\}{:}y$. Since the tableau is completed, the minimum number of unsatisfied formulas is 1.*

6.4.1 Soundness and Completeness

We now prove the soundness and completeness:

Lemma 6.1 *The construction of a completed Signed MaxSAT tableau for a multiset of signed formulas takes a finite time.*

Proof 6.4 *The termination follows from the fact that the application of the α, β and $\neg$-rules reduces the number of connectives, and the application of the $\square$-rule reduces the number of signed literals.*

Lemma 6.2 *Any assignment I unsatisfies the same number of literals in the multisets of signed literals $\{S_1{:}x, S_2{:}x\}$ and $\{(S_1 \cap S_2){:}x, (S_1 \cup S_2){:}x\}$.*

Proof 6.5 *we distinguish four cases: i) I satisfies $S_1{:}x$ and $S_2{:}x$: Since $I(x) \in S_1$ and $I(x) \in S_2$, then I trivially satisfies $(S_1 \cap S_2){:}x$ and $(S_1 \cup S_2){:}x$; ii) I unsatisfies $S_1{:}x$ and satisfies $S_2{:}x$: Since $I(x) \notin S_1$ and $I(x) \in S_2$, then I trivially unsatisfies $(S_1 \cap S_2){:}x$ and satisfies $(S_1 \cup S_2){:}x$; iii) I unsatisfies $S_2{:}x$ and satisfies $S_1{:}x$: Since $I(x) \in S_1$ and $I(x) \notin S_2$, then I trivially unsatisfies $(S_1 \cap S_2){:}x$ and satisfies $(S_1 \cup S_2){:}x$ iv) I unsatisfies $S_1{:}x$ and $S_2{:}x$: Since $I(x) \notin S_1$ and $I(x) \notin S_2$, I assigns to x a truth value not belonging neither to S_1 nor S_2 and unsatisfies $(S_1 \cap S_2){:}x$ and $(S_1 \cup S_2){:}x$. Therefore, the number of unsatisfied literals is preserved.*

Theorem 6.2 *The Signed MaxSAT tableau calculus is sound.*

Proof 6.6 *Since signed MaxSAT tableaux are derived after a finite number of applications of the expansion rules of Table 6.5, it is enough to prove that these rules are sound. The soundness of the α, β and $\neg$-rules follows from the soundness in the Boolean case. Thus, we only prove the soundness of the $\square$-rule: If there are m premises, we obtain the conclusion by applying $m-1$ times Lemma 6.2. In the first application, we replace the first two premises by $\{(S_1 \cap S_2){:}x, (S_1 \cup S_2){:}x\}$. In the second application, we apply the lemma to $\{(S_1 \cap S_2){:}x, (S_3){:}x\}$ and replace these two signed literals by $\{(S_1 \cap S_2 \cap S_3){:}x, (S_1 \cap S_2) \cup (S_3){:}x\}$. In the i-th application we we apply the lemma to $\{(S_1 \cap \cdots \cap S_{i-1}){:}x, (S_i){:}x\}$ and replace these two signed literals by $\{(S_1 \cap \cdots \cap S_i){:}x, (S_1 \cap \cdots \cap S_{i-1}) \cup (S_i){:}x\}$. Since $S_1 \cap \cdots \cap S_m = \emptyset$, in the last step, we derive a contradiction from $(S_1 \cap \cdots \cap S_m){:}x = \square$ and add the compensation literals.*

Theorem 6.3 *The Signed MaxSAT tableau calculus is complete.*

Proof 6.7 *Assume that there is a completed tableau T for ϕ that does not have cost m. We distinguish two cases:*

(i) T has a branch b of cost k, where $k < m$. Then, T has a branch with k contradictions and a satisfiable multiset of signed literals because T is completed. This implies that the minimum number of unsatisfied active formulas among the branches of T is at most k. Since the completed tableau was derived from the initial tableau by applying sound inference rules, this also holds for the initial tableau. However, this is in contradiction with m being the minimum number of unsatisfied formulas in ϕ because $k < m$. Thus, any branch of T has at least cost m.

(ii) T has no branch of cost m. This is in contradiction with m being the minimum number of unsatisfied formulas in ϕ. Since the tableau rules preserve the minimum

number of unsatisfied formulas and the branches of any completed tableau only contain contradictions and active formulas that form a satisfiable multiset of signed literals, T must have a saturated branch with m contradictions. Thus, T has a branch of cost m.

Hence, each completed tableau for a multiset of signed formulas ϕ has cost m if the minimum number of formulas that can be unsatisfied in ϕ is m.

6.4.2 Extending to the Weighted Partial Signed MaxSAT

Table 6.6 shows the expansion rules for Weighted Partial Signed MaxSAT, where the hard premises have weight $\top$. The α, β and $\neg$-rule are the same as in SAT when all their premises are hard. The $\square$-rule derives $\blacksquare$ when a contradiction is derived when all the premises are hard; in this case, an unfeasible solution has been detected and the search stops in the current branch. For weighted soft clauses, the α, β and $\neg$-rule have just one premise and the weight associated with the premise is transferred to the conclusions. If the $\square$-rule has soft premises with different weights, we split the premises into two groups: Given a clause (C_i, w_i), one group has (C_i, w) with weight $w = \min(w_{k+1}, \cdots, w_m)$ and the other group has $(C_i, w_i - w)$ (if $w_i - w = 0$, the clause is not added to the second group). The clauses of this second group are added as compensations clauses and can be used to detect further contradictions. In this way, since the first group has clauses with identical weights, we soundly derive a contradiction with weight w. Hard premises remain active and can be used to detect further contradictions.

In the weighted case, when a branch has repeated occurrences of a formula A, say $(A, w_1), \ldots, (A, w_s)$, such occurrences can be replaced with the single formula $(A, w_1 + \cdots + w_s)$. Moreover, the cost of a saturated weighted branch is the sum of the weights of the boxes that appear in the branch, and the cost of a completed weighted tableau is the minimum cost among all its branches. We claim that the proposed calculus is sound and complete.

6.5 Conclusion

Table 6.6: Tableau expansion rules for Weighted Partial Signed MaxSAT

$$\frac{(\alpha, \top)}{(\alpha_1, \top)\quad (\alpha_2, \top)}$$

$$\frac{(\alpha, w)}{\begin{array}{c|c}(\Box, w) & (\alpha_1, w)\\ & (\alpha_2, w)\end{array}}$$

α-rule

$$\frac{(\beta, \top)}{\begin{array}{c|c}(\beta_1, \top) & (\beta_2, \top)\end{array}}$$

$$\frac{(\beta, w)}{\begin{array}{c|c}(\beta_1, w) & (\beta_2, w)\end{array}}$$

β-rule

$$\frac{(\neg(\neg A), \top)}{(A, \top)}\qquad \frac{(\neg(\neg A), w)}{(A, w)}$$

$\neg$-rule

$$\frac{\begin{array}{c}(S_1{:}x, \top)\\ \dots\\ (S_m{:}x, \top)\end{array}}{\blacksquare}$$

$\Box$-rule

$$\frac{\begin{array}{c}(S_1{:}x, w_1)\\ \dots\\ (S_m{:}x, w_m)\end{array}}{\begin{array}{c}(\Box, w)\\ ((S_1 \cup S_2){:}x, w)\\ \dots\dots\dots\\ ((S_1 \cap \dots \cap S_{m-1}) \cup S_m{:}x, w)\\ (S_1{:}x, w_1 - w)\\ \dots\dots\dots\\ (S_m{:}x, w_m - w)\end{array}}$$

provided that $w = \min(w_1, \cdots, w_m)$, and $S_1 \cap \dots \cap S_m = \emptyset$

$$\frac{\begin{array}{c}(S_1{:}x, \top)\\ \dots\\ (S_k{:}x, \top)\\ (S_{k+1}{:}x, w_{k+1})\\ \dots\\ (S_m{:}x, w_m)\end{array}}{\begin{array}{c}(\Box, w)\\ (S_1{:}x, \top)\\ \dots\\ (S_k{:}x, \top)\\ ((S_1 \cap \dots \cap S_k) \cup S_{k+1}{:}x, w)\\ \dots\dots\dots\\ ((S_1 \cap \dots \cap S_{m-1}) \cup S_m{:}x, w)\\ (S_{k+1}{:}x, w_{k+1} - w)\\ \dots\dots\dots\\ (S_m{:}x, w_m - w)\end{array}}$$

provided that $w = \min(w_{k+1}, \cdots, w_m)$, $k \geq 1$ and $S_1 \cap \dots \cap S_m = \emptyset$

$\Box$-rule

Conclusion

This manuscript is composed by two parts. In the background part, we did a detailed review around the two major logic problem: the satisfiabilty problem (SAT) and its optimization extension maximum satisfiabilty problem (MaxSAT). We introduced their definitions and variants, then focused on the solving techniques in state of the art SAT and MaxSAT solvers. In the contribution part, we *****

First, we introduced a novel method to improve the performance of SAT solvers by guiding formula simplifications using variable heuristic scores. This method effectively integrates solution search and formula simplifications, enhancing overall efficiency. The heuristic-based approach allows the SAT solver to better prioritize variable elimination, resulting in improved solver performance in practical applications.

Second, for MaxSAT solvers, the work advances the field through the enhancement of the lower bound estimation in Branch and Bound solvers by utilizing an unlocking mechanism. This mechanism accelerates the pruning process, which leads to faster convergence toward optimal solutions. The improved estimation techniques help in addressing larger, more complex instances of MaxSAT, significantly improving solver performance in real-world optimization problems.

Finally, the work makes notable contributions to the theoretical foundations of MaxSAT by improving the existing MaxSAT Tableau calculus. The introduction of new inference rules and the extension of the calculus to support signed MaxSAT provide a richer theoretical framework. This advancement broadens the applicability of the proof system, offering a more flexible and powerful tool for solving MaxSAT problems, especially in domains that require signed logic and more complex logical constructs

Bibliography

[1] André Abramé and Djamal Habet. “Ahmaxsat: Description and Evaluation of a Branch and Bound Max-SAT Solver”. In: *J. Satisf. Boolean Model. Comput.* (2014), pp. 89–128 (cit. on p. 54).

[2] André Abramé and Djamal Habet. “Efficient application of max-sat resolution on inconsistent subsets”. In: *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*. Springer. 2014, pp. 92–107 (cit. on pp. 57, 97).

[3] David Allouche, Isabelle André, Sophie Barbe, et al. “Computational protein design as an optimization problem”. In: *Artificial Intelligence* 212 (2014), pp. 59–79 (cit. on p. 53).

[4] Teresa Alsinet. “Improved branch and bound algorithms for Max-SAT”. In: *Sixth International Conference on Theory and Applications of Satisfiability Testing, 2003*. 2003 (cit. on p. 56).

[5] Mario Alviano, Carmine Dodaro, and Francesco Ricca. “A MaxSAT algorithm using cardinality constraints of bounded size”. In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015 (cit. on p. 62).

[6] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. “Solving (weighted) partial MaxSAT through satisfiability testing”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2009, pp. 427–440 (cit. on p. 62).

[7] Carlos Ansótegui, Frédéric Didier, and Joel Gabàs. “Exploiting the structure of unsatisfiable cores in MaxSAT”. In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015 (cit. on p. 62).

[8] Carlos Ansótegui, Felip Manyà, Jesus Ojeda, et al. “Incomplete MaxSAT approaches for combinatorial testing”. In: *Journal of Heuristics* 28.4 (2022), pp. 377–431 (cit. on p. 53).

[9] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, et al. “Boosting local search thanks to CDCL”. In: *Logic for Programming, Artificial Intelligence, and Reasoning: 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings 17*. Springer. 2010, pp. 474–488 (cit. on p. 44).

[10] Fahiem Bacchus, Matti Järvisalo, and Martins Ruben. “Maximum Satisfiability”. In: *Handbook of satisfiability, second edition*. IOS Press, 2021, pp. 929–991 (cit. on p. 14).

[11] Adrian Balint and Andreas Fröhlich. “Improving stochastic local search for SAT with a new probability distribution”. In: *Theory and Applications of Satisfiability Testing–SAT 2010: 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 13*. Springer. 2010, pp. 10–15 (cit. on p. 46).

[12] Adrian Balint and Uwe Schöning. “Choosing probability distributions for stochastic local search and the role of make versus break”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2012, pp. 16–29 (cit. on pp. 44, 46).

[13] Paul Beame, Henry Kautz, and Ashish Sabharwal. “Understanding the power of clause learning”. In: *IJCAI*. Citeseer. 2003, pp. 1194–1201 (cit. on p. 70).

[14] Bernhard Beckert, Reiner Hähnle, and Felip Manyà. “The SAT problem of signed CNF formulas”. In: *Labelled Deduction* (2000), pp. 59–80 (cit. on p. 100).

[15] 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 (cit. on p. 63).

[16] Jeremias Berg, Emir Demirovic, and Peter Stuckey. “Loandra in the 2020 maxsat evaluation”. In: *MaxSAT Evaluation* 2020 (2020), p. 10 (cit. on p. 62).

[17] Jeremias Berg, Emir Demirovic, and Peter J. Stuckey. “Core-Boosted Linear Search for Incomplete MaxSAT”. In: *Proceedings of CPAIOR 2019*. 2019, pp. 39–56.

[18] Merrie Bergmann, James Moor, Jack Nelson, et al. *The logic book*. Vol. 3. McGraw-Hill New York, 1998 (cit. on p. 20).

[19] A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability: Second Edition*. IOS Press, 2021 (cit. on pp. 14, 25).

[20] Armin Biere. “Adaptive restart strategies for conflict driven SAT solvers”. In: *Theory and Applications of Satisfiability Testing–SAT 2008: 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings 11*. Springer. 2008, pp. 28–33 (cit. on p. 40).

[21] Armin Biere. “Lingeling, plingeling, picosat and precosat at sat race 2010”. In: (2010) (cit. on pp. 35, 40).

[22] Armin Biere. “Yet another local search solver and Lingeling and friends entering the SAT Competition 2014”. In: *Sat competition* 2014.2 (2014), p. 65 (cit. on p. 44).

[23] Armin Biere. “CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017”. In: *Proc. of SAT Competition 2017: Solver and Benchmark Descriptions*. University of Helsinki, 2017, pp. 14–15 (cit. on pp. 25, 35, 46, 73).

[24] Armin Biere. “CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018”. In: *Proc. of SAT Competition 2018: Solver and Benchmark Descriptions*. University of Helsinki, 2018, pp. 13–14 (cit. on p. 46).

[25] Armin Biere, Alessandro Cimatti, Edmund Clarke, et al. “Symbolic model checking without BDDs”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*. Springer. 1999, pp. 193–207 (cit. on p. 25).

[26] Armin Biere, Katalin Fazekas, Mathias Fleury, et al. “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020”. In: *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*. University of Helsinki, 2020, pp. 51–53 (cit. on pp. 25, 35, 46, 48, 69, 79).

[27] Armin Biere and Andreas Fröhlich. “Evaluating CDCL Variable Scoring Schemes”. In: *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing, SAT*. 2015, pp. 405–422 (cit. on p. 73).

[28] Armin Biere and Andreas Fröhlich. “Evaluating CDCL variable scoring schemes”. In: *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer. 2015, pp. 405–422 (cit. on pp. 40, 42).

[29] Miquel Bofill, Jordi Coll, Marc Garcia, et al. “Constraint solving approaches to the business-to-business meeting scheduling problem”. In: *Journal of Artificial Intelligence Research* 74 (2022), pp. 263–301 (cit. on p. 53).

[30] María Luisa Bonet, Jordi Levy, and Felip Manyà. “Resolution for max-sat”. In: *Artificial Intelligence* 171.8-9 (2007), pp. 606–618 (cit. on pp. 57, 97).

[31] Brian Borchers and Judith Furman. “A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems”. In: *Journal of Combinatorial Optimization* 2 (1998), pp. 299–306 (cit. on pp. 53, 54, 56).

[32] Joshua Brakensiek, Marijn Heule, John Mackey, et al. “The resolution of Keller’s conjecture”. In: *Journal of Automated Reasoning* 66.3 (2022), pp. 277–300 (cit. on p. 25).

[33] Curtis Bright, Kevin KH Cheung, Brett Stevens, et al. “A SAT-based Resolution of Lam’s Problem”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 3669–3676 (cit. on p. 25).

[34] Shaowei Cai. “Balance between complexity and quality: Local search for minimum vertex cover in massive graphs”. In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015 (cit. on p. 65).

[35] 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 (cit. on p. 66).

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

[37] Shaowei Cai, Chuan Luo, and Kaile Su. “CCASat: solver description”. In: *Proceedings of SAT challenge* 2012 (2012), p. 13 (cit. on p. 45).

[38] Shaowei Cai, Chuan Luo, and Kaile Su. “CCAnr: A configuration checking based local search solver for non-random satisfiability”. In: *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer. 2015, pp. 1–8 (cit. on pp. 44, 46).

[39] Shaowei Cai, Chuan Luo, John Thornton, et al. “Tailoring local search for partial MaxSAT”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 28. 1. 2014 (cit. on pp. 64, 66).

[40] Shaowei Cai, Chuan Luo, and Haochen Zhang. “From Decimation to Local Search and Back: A New Approach to MaxSAT.” In: *IJCAI*. 2017, pp. 571–577 (cit. on p. 64).

[41] Shaowei Cai, Chuan Luo, Xindi Zhang, et al. “Improving local search for structured SAT formulas via unit propagation based construct and cut initialization (short paper)”. In: *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2021 (cit. on p. 44).

[42] Shaowei Cai and Kaile Su. “Local search with configuration checking for SAT”. In: *2011 IEEE 23rd international conference on tools with artificial intelligence*. IEEE. 2011, pp. 59–66.

[43] Shaowei Cai, Kaile Su, and Abdul Sattar. “Local search with edge weighting and configuration checking heuristics for minimum vertex cover”. In: *Artificial Intelligence* 175.9-10 (2011), pp. 1672–1696 (cit. on p. 46).

[44] Shaowei Cai, Kaile Su, and Abdul Sattar. “Local search with edge weighting and configuration checking heuristics for minimum vertex cover”. In: *Artificial Intelligence* 175.9-10 (2011), pp. 1672–1696 (cit. on p. 65).

[45] Byungki Cha, Kazuo Iwama, Yahiko Kambayashi, et al. “Local search algorithms for partial MAXSAT”. In: *AAAI/IAAI* 263268 (1997), p. 9 (cit. on pp. 65, 66).

[46] Jiejiang Chen, Shaowei Cai, Yiyuan Wang, et al. “Improved local search for the minimum weight dominating set problem in massive graphs by using a deep optimization mechanism”. In: *Artificial Intelligence* 314 (2023), p. 103819 (cit. on p. 65).

[47] Zhihan Chen, Xindi Zhang, Shaowei Cai, et al. “CDCL Solvers with Improved Local Search Cooperation and Pre-processing”. In: *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*. University of Helsinki, 2022, pp. 37–38 (cit. on p. 73).

[48] Mohamed Sami Cherif, Djamal Habet, and André Abramé. “Understanding the power of Max-SAT resolution through UP-resilience”. In: *Artificial Intelligence* (2020), p. 103397 (cit. on p. 54).

[49] Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. “Combining VSIDS and CHB Using Restarts in SAT”. In: *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming, CP*. 2021, 20:1–20:19 (cit. on p. 73).

[50] Yi Chu, Shaowai Cai, Zhendong Lei, et al. “Nuwls-c: Solver description”. In: *MaxSAT Evaluation* 2022 (2022), p. 28 (cit. on p. 67).

[51] Yi Chu, Shaowei Cai, and Chuan Luo. “Nuwls-c-2023: Solver description”. In: *MaxSAT Evaluation 2023* (2023), p. 23 (cit. on pp. 63, 66, 67).

[52] Yi Chu, Shaowei Cai, and Chuan Luo. “NuWLS: Improving local search for (weighted) partial MaxSAT by new weighting techniques”. In: *Proceedings of AAAI 2023*. Vol. 37. 2023, pp. 3915–3923 (cit. on pp. 63–66).

[53] Lorenzo Ciampiconi, Bishwamittra Ghosh, Jonathan Scarlett, et al. “A MaxSAT-based framework for group testing”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 06. 2020, pp. 10144–10152 (cit. on p. 53).

[54] Alan Cobham. “The intrinsic computational difficulty of functions”. In: (1965) (cit. on p. 19).

[55] Jordi Coll, Shuolin Li, Chu-Min Li, et al. “MaxCDCL and WMaxCDCL in MaxSAT Evaluation 2022”. In: *MaxSAT Evaluation 2022* (2022), pp. 15–16 (cit. on p. 63).

[56] Jordi Coll, Shuolin Li, Chu-Min Li, et al. “WMaxCDCL in MaxSAT Evaluation 2023”. In: *MaxSAT Evaluation 2023* (2023), pp. 16–17 (cit. on p. 63).

[57] Jordi Coll, Shuolin Li, Chu-Min Li, et al. “WMaxCDCL in MaxSAT Evaluation 2023”. In: *MaxSAT Evaluation 2023* (2023), pp. 16–17 (cit. on p. 15).

[58] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. ACM, 1971, pp. 151–158 (cit. on pp. 14, 19, 24, 25).

[59] Marcello d’Agostino et al. “Tableau methods for classical propositional logic”. In: *Handbook of tableau methods* (1999), pp. 45–123 (cit. on p. 98).

[60] Dominique d’Almeida and Éric Grégoire. “Model-based diagnosis with default information implemented through MAX-SAT technology”. In: *2012 IEEE 13th International Conference on Information Reuse & Integration (IRI)*. IEEE. 2012, pp. 33–36 (cit. on p. 53).

[61] Martin Davis, George Logemann, and Donald Loveland. “A machine program for theorem-proving”. In: *Communications of the ACM* 5.7 (1962), pp. 394–397 (cit. on pp. 14, 26, 29).

[62] Martin Davis and Hilary Putnam. “A computing procedure for quantification theory”. In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215 (cit. on pp. 26, 47).

[63] Emir Demirović, Nysret Musliu, and Felix Winter. “Modeling and solving staff scheduling with partial weighted maxSAT”. In: *Annals of Operations Research* 275 (2019), pp. 79–99 (cit. on p. 53).

[64] Jack Edmonds. “Paths, trees, and flowers”. In: *Canadian Journal of mathematics* 17 (1965), pp. 449–467 (cit. on p. 19).

[65] Niklas Eén and Armin Biere. “Effective preprocessing in SAT through variable and clause elimination”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2005, pp. 61–75 (cit. on p. 48).

[66] Niklas Eén, Alan Mishchenko, and Niklas Sörensson. “Applying logic synthesis for speeding up SAT”. In: *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings 10*. Springer. 2007, pp. 272–286 (cit. on p. 25).

[67] 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 6*. Springer. 2004, pp. 502–518 (cit. on pp. 25, 35, 40, 46, 79).

[68] Guido Fiorino. “New tableau characterizations for non-clausal MaxSAT problem”. In: *Logic Journal of the IGPL* 30.3 (2022), pp. 422–436 (cit. on pp. 103, 106).

[69] Jeremy Frank, P Cheeseman, and J Allen. “Weighting for Godot: Learning heuristics for GSAT”. In: *Proceedings of the National Conference on Artificial Intelligence*. Vol. 13. 1996, pp. 338–343 (cit. on p. 66).

[70] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. University of Pennsylvania, 1995 (cit. on p. 55).

[71] Zhaohui Fu and Sharad Malik. “On Solving the Partial MAX-SAT Problem”. In: *Proceedings of SAT 2006*. 2006, pp. 252–265 (cit. on p. 53).

[72] Zhaohui Fu and Sharad Malik. “On solving the partial MAX-SAT problem”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2006, pp. 252–265 (cit. on p. 62).

[73] Oliver Gableske and Marijn JH Heule. “EagleUP: Solving random 3-SAT using SLS with unit propagation”. In: *Theory and Applications of Satisfiability Testing-SAT 2011: 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings 14*. Springer. 2011, pp. 367–368 (cit. on p. 45).

[74] Ian P Gent, Toby Walsh, et al. “Towards an understanding of hill-climbing procedures for SAT”. In: *AAAI*. Vol. 93. Citeseer. 1993, pp. 28–33 (cit. on p. 45).

[75] Fred Glover. “Tabu search: A tutorial”. In: *Interfaces* 20.4 (1990), pp. 74–94 (cit. on p. 65).

[76] Andreas Goerdt. “Regular Resolution Versus Unrestricted Resolution”. In: *SIAM Journal on Computing* 22.4 (1993), pp. 661–683 (cit. on p. 70).

[77] Federico Heras and Javier Larrosa. “New inference rules for efficient Max-SAT solving”. In: *AAAI*. 2006, pp. 68–73 (cit. on pp. 57, 97).

[78] Federico Heras, Javier Larrosa, and Albert Oliveras. “MiniMaxSat: A new weighted Max-SAT solver”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2007, pp. 41–55 (cit. on pp. 54, 56).

[79] Federico Heras, Javier Larrosa, and Albert Oliveras. “MiniMaxSAT: An Efficient Weighted Max-SAT Solver”. In: *Journal of Artificial Intelligence Research* (2008), pp. 1–32.

[80] Federico Heras, Javier Larrosa, and Albert Oliveras. “MiniMaxSAT: An efficient weighted Max-SAT solver”. In: *Journal of Artificial Intelligence Research* 31 (2008), pp. 1–32 (cit. on pp. 54, 56).

[81] Marijn Heule. “Schur number five”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018 (cit. on p. 25).

[82] Marijn JH Heule, Oliver Kullmann, and Victor W Marek. “Solving and verifying the boolean pythagorean triples problem via cube-and-conquer”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2016, pp. 228–245 (cit. on p. 25).

[83] Holger H Hoos and Thomas Stv̈tzle. “Stochastic local search”. In: *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2018, pp. 297–307 (cit. on p. 43).

[84] Wenqui Huang and Xiangdong Yu. “A DNF without Regular Shortest Consensus Path”. In: *SIAM J. Comput.* 16.5 (1987), pp. 836–840 (cit. on p. 70).

[85] Frank Hutter, Dave AD Tompkins, and Holger H Hoos. “Scaling and probabilistic smoothing: Efficient dynamic local search for SAT”. In: *Principles and Practice of Constraint Programming-CP 2002: 8th International Conference, CP 2002 Ithaca, NY, USA, September 9–13, 2002 Proceedings 8*. Springer. 2002, pp. 233–248 (cit. on p. 45).

[86] Saıd Jabbour, Nizar Mhadhbi, Badran Raddaoui, et al. “A SAT-based framework for overlapping community detection in networks”. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2017, pp. 786–798 (cit. on p. 53).

[87] Robert G Jeroslow and Jinchang Wang. “Solving propositional satisfiability problems”. In: *Annals of mathematics and Artificial Intelligence* 1.1 (1990), pp. 167–187 (cit. on p. 55).

[88] Richard M. Karp. “Reducibility Among Combinatorial Problems”. In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Plenum Press, New York, 1972, pp. 85–103 (cit. on p. 25).

[89] Miyuki Koshimura, Hidetomo Nabeshima, Hiroshi Fujita, et al. “Solving open job-shop scheduling problems by SAT encoding”. In: *IEICE TRANSACTIONS on Information and Systems* 93.8 (2010), pp. 2316–2318 (cit. on p. 25).

[90] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, et al. “QMaxSAT: A partial Max-SAT solver”. In: *Journal on Satisfiability, Boolean Modeling and Computation* (2012), pp. 95–100 (cit. on p. 62).

[91] Adrian Kuegel. “Improved Exact Solver for the Weighted MAX-SAT Problem”. In: *Proceedings of Workshop Pragmatics of SAT, POS-10, Edinburgh, UK*. 2010, pp. 15–27 (cit. on p. 54).

[92] Andreas Kuehlmann and Florian Krohm. “Equivalence checking using cuts and heaps”. In: *Proceedings of the 34th annual Design Automation Conference*. 1997, pp. 263–268 (cit. on p. 25).

[93] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, et al. “Robust Boolean reasoning for equivalence checking and functional property verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.12 (2002), pp. 1377–1394 (cit. on p. 25).

[94] A. H. Land and A. G. Doig. “An Automatic Method of Solving Discrete Programming Problems”. In: *Econometrica* 28.3 (1960), pp. 497–520.

[95] Javier Larrosa, Federico Heras, and Simon De Givry. “A logical approach to efficient Max-SAT solving”. In: *Artificial Intelligence* 172.2-3 (2008), pp. 204–233 (cit. on pp. 57, 97).

[96] 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 (cit. on p. 62).

[97] Zhendong Lei and Shaowei Cai. “Solving (Weighted) Partial MaxSAT by Dynamic Local Search for SAT”. In: *Proceedings of IJCAI 2018*. 2018, pp. 1346–1352 (cit. on pp. 63–66).

[98] Zhendong Lei, Shaowei Cai, Fei Geng, et al. “SATLike-c: Solver description”. In: *MaxSAT Evaluation* 2021 (2021), p. 19 (cit. on p. 67).

[99] Chu Min Li and Wen Qi Huang. “Diversification and determinism in local search for satisfiability”. In: *Theory and Applications of Satisfiability Testing: 8th International Conference, SAT 2005, St Andrews, UK, June 19-23, 2005. Proceedings 8*. Springer. 2005, pp. 158–172 (cit. on p. 46).

[100] Chu Min Li and Yu Li. “Satisfying versus falsifying in local search for satisfiability”. In: *Theory and Applications of Satisfiability Testing–SAT 2012: 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings 15*. Springer. 2012, pp. 477–478 (cit. on pp. 44–46).

[101] Chu Min Li and Felip Manyà. “MaxSAT, Hard and Soft Constraints”. In: *Handbook of Satisfiability, second edition*. IOS Press, 2021, pp. 903–927 (cit. on p. 14).

[102] Chu Min Li, Felip Manyà, Nouredine Mohamedou, et al. “Exploiting cycle structures in Max-SAT”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2009, pp. 467–480 (cit. on p. 54).

[103] Chu Min Li, Felip Manyà, Nouredine Ould Mohamedou, et al. “Transforming inconsistent subformulas in MaxSAT lower bound computation”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2008, pp. 582–587 (cit. on p. 54).

[104] Chu Min Li, Felip Manyà, Nouredine Ould Mohamedou, et al. “Resolution-based lower bounds in MaxSAT”. In: *Constraints* 15.4 (2010), pp. 456–484 (cit. on pp. 54, 56).

[105] Chu Min Li, Felip Manyà, and Jordi Planes. “Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers”. In: *International conference on principles and practice of constraint programming*. Springer. 2005, pp. 403–414 (cit. on p. 53).

[106] Chu Min Li, Felip Manyà, and Jordi Planes. “New Inference Rules for Max-SAT”. In: *Journal of Artificial Intelligence Research* (2007), pp. 321–359 (cit. on p. 54).

[107] Chu Min Li, Felip Manyà, and Jordi Planes. “New inference rules for Max-SAT”. In: *Journal of Artificial Intelligence Research* 30 (2007), pp. 321–359 (cit. on pp. 56, 57, 97).

[108] Chu Min Li, Felip Manyà, and Joan Ramon Soler. “A Clause Tableau Calculus for MaxSAT.” In: *IJCAI*. 2016, pp. 766–772 (cit. on pp. 98, 101, 103, 106).

[109] Chu Min Li, Felip Manyà, and Joan Ramon Soler. “A tableau calculus for non-clausal maximum satisfiability”. In: *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer. 2019, pp. 58–73 (cit. on pp. 98, 103, 106).

[110] Chu Min Li and Zhe Quan. “An Efficient Branch-and-Bound Algorithm Based on MaxSAT for the Maximum Clique Problem”. In: *Proceedings of AAAI 2010*. 2010, pp. 128–133 (cit. on p. 53).

[111] Chu Min Li, Wanxia Wei, and Harry Zhang. “Combining adaptive noise and look-ahead in local search for SAT”. In: *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings 10*. Springer. 2007, pp. 121–133 (cit. on p. 46).

[112] Chu-Min Li, Jordi Coll, Shuolin Li, et al. “MaxCDCL in MaxSAT Evaluation 2023”. In: *MaxSAT Evaluation 2023* (2023), pp. 14–15 (cit. on p. 15).

[113] Chu-Min Li, Shuolin Li, Jordi Coll, et al. “MaxCDCL in MaxSAT Evaluation 2024”. In: *MaxSAT Evaluation 2024* (2024), pp. 15–16 (cit. on p. 15).

[114] Chu-Min Li, Zhenxing Xu, Jordi Coll, et al. “Boosting branch-and-bound MaxSAT solvers with clause learning”. In: *AI Communications* (2021), pp. 1–21 (cit. on pp. 54, 56).

[115] Chu-Min Li, Zhenxing Xu, Jordi Coll, et al. “Combining clause learning and branch and bound for MaxSAT”. In: *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. 2021 (cit. on pp. 54, 56).

[116] Shoulin Li, Jordi Coll, Djamal Habet, et al. “A tableau calculus for maxsat based on resolution”. In: *Artificial Intelligence Research and Development*. IOS Press, 2022, pp. 35–44 (cit. on p. 15).

[117] Shuolin Li. “Guiding Problem Simplification with Variable Activities in SAT Solvers”. In: *Doctoral Consortium at ECAI-2024*. Doctoral Consortium paper. 2024 (cit. on p. 15).

[118] Shuolin Li, Jordi Coll, Djamal Habet, et al. “A Tableau Calculus for Signed Maximum Satisfiability”. In: *2023 IEEE 53rd International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE. 2023, pp. 170–175 (cit. on p. 15).

[119] Shuolin Li, Jordi Coll, Chu-Min Li, et al. “Solvers Cadical ESA and Kissat MAB ESA in 2022 SAT competition”. In: *SAT COMPETITION* 2022 (2022), pp. 33–34 (cit. on p. 15).

[120] Shuolin Li, Chu-Min Li, Jordi Coll, et al. “Get the Better Cores through Unlocking Mechanism in Unit Propagation”. In: *Doctoral Program of the International Conference on Principles and Practice of Constraint Programming (CP)*. Doctoral Program paper. 2024 (cit. on p. 15).

[121] Shuolin Li, Chu-Min Li, Jordi Coll, et al. “WMaxCDCL in MaxSAT Evaluation 2024”. In: *MaxSAT Evaluation 2024* (2024), pp. 17–18 (cit. on p. 15).

[122] Shuolin Li, Chu-Min Li, Mao Luo, et al. “A new variable ordering for in-processing bounded variable elimination in sat solvers”. In: *32th International Joint Conferences on Artificial Intelligence (IJCAI 2023)*. 2023 (cit. on p. 79).

[123] Shuolin Li, Chu-Min Li, Mao Luo, et al. “A new variable ordering for in-processing bounded variable elimination in sat solvers”. In: *32th International Joint Conferences on Artificial Intelligence (IJCAI 2023)*. 2023 (cit. on p. 15).

[124] Shuolin Li, Chu-Min Li, Mao Luo, et al. “ESA Solvers, Kissat MAB Binary and AMSAT in SAT Competition 2023”. In: *SAT COMPETITION* 2023 (2023), pp. 32–33 (cit. on p. 15).

[125] Shuolin Li, Chu-Min Li, Mao Luo, et al. “ESA Solvers, Kissat MAB Binary and AMSAT in SAT Competition 2024”. In: *SAT COMPETITION* 2024 (2024), pp. 15–16 (cit. on p. 15).

[126] Yih-Lang Li, Shih-Ting Lin, Shinichi Nishizawa, et al. “Mcell: multi-row cell layout synthesis with resource constrained max-sat based detailed routing”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–8 (cit. on p. 53).

[127] Jia Liang, Vijay Ganesh, Pascal Poupart, et al. “Exponential recency weighted average branching heuristic for SAT solvers”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1. 2016 (cit. on pp. 35, 41).

[128] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, et al. “Exponential Recency Weighted Average Branching Heuristic for SAT Solvers”. In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence, AAAI*. 2016, pp. 3434–3440 (cit. on p. 73).

[129] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, et al. “Learning rate based branching heuristic for SAT solvers”. In: *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19*. Springer. 2016, pp. 123–140 (cit. on pp. 35, 41).

[130] Jia Hui Liang, Chanseok Oh, Vijay Ganesh, et al. “Maple-comsps, maplecomsps lrb, maplecomsps chb”. In: *Proceedings of SAT Competition* 2016 (2016) (cit. on pp. 35, 41).

[131] Chuan Luo, Shaowei Cai, Kaile Su, et al. “Clause states based configuration checking in local search for satisfiability”. In: *IEEE transactions on Cybernetics* 45.5 (2014), pp. 1028–1041 (cit. on p. 46).

[132] Chuan Luo, Shaowei Cai, Kaile Su, et al. “CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability”. In: *Artificial Intelligence* 243 (2017), pp. 26–44.

[133] Chuan Luo, Shaowei Cai, Kaile Su, et al. “CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability”. In: *Artificial Intelligence* 243 (2017), pp. 26–44 (cit. on pp. 64–66).

[134] Chuan Luo, Shaowei Cai, Wei Wu, et al. “CCLS: an efficient local search algorithm for weighted maximum satisfiability”. In: *IEEE Transactions on Computers* 64.7 (2014), pp. 1830–1843 (cit. on p. 65).

[135] Mao Luo, Chu-Min Li, Xinyun Wu, et al. “Branching Strategy Selection Approach Based on Vivification Ratio”. In: *arXiv preprint arXiv:2112.06917* (2021) (cit. on p. 15).

[136] Yogesh S Mahajan, Zhaohui Fu, and Sharad Malik. “Zchaff2004: An efficient sat solver”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2004, pp. 360–375 (cit. on p. 35).

[137] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. “Algorithms for weighted boolean optimization”. In: *Theory and Applications of Satisfiability Testing-SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30-July 3, 2009. Proceedings 12*. Springer. 2009, pp. 495–508 (cit. on p. 62).

[138] Norbert Manthey. “The MergeSat Solver”. In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu-Min Li and Felip Manyà. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 387–398 (cit. on p. 74).

[139] Felip Manyà, Santiago Negrete, Carme Roig, et al. “Solving the team composition problem in a classroom”. In: *Fundamenta Informaticae* 174.1 (2020), pp. 83–101 (cit. on p. 53).

[140] Joao Marques-Silva. “Search algorithms for satisfiability problems in combinational switching circuits”. PhD thesis. University of Michigan, 1995 (cit. on pp. 14, 35).

[141] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. “Boosting complete techniques thanks to local search methods”. In: *Annals of mathematics and artificial intelligence* 22 (1998), pp. 319–331 (cit. on p. 44).

[142] David McAllester, Bart Selman, Henry Kautz, et al. “Evidence for invariants in local search”. In: *AAAI/IAAI*. Rhode Island, USA. 1997, pp. 321–326 (cit. on p. 46).

[143] Wil Michiels, Emile Aarts, and Jan Korst. *Theoretical aspects of local search*. Vol. 25. Springer, 2007 (cit. on p. 46).

[144] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, et al. “Chaff: Engineering an efficient SAT solver”. In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 530–535 (cit. on p. 35).

[145] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, et al. “Chaff: Engineering an Efficient SAT Solver”. In: *Proceedings of the 38th Design Automation Conference, DAC*. 2001, pp. 530–535 (cit. on pp. 26, 39, 73).

[146] Hidetomo Nabeshima, Koji Iwanuma, and Katsumi Inoue. “GLUEMINISAT 2.2. 10 & 2.2. 10-5”. In: 2015 (cit. on p. 48).

[147] Alexander Nadel. “Polarity and variable selection heuristics for SAT-based anytime MaxSAT”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 12.1 (2020), pp. 17–22.

[148] Alexander Nadel. “Tt-open-wbo-inc-20: an anytime maxsat solver entering mse’20”. In: *MaxSAT Evaluation* (2020), pp. 32–33 (cit. on p. 62).

[149] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994 (cit. on p. 52).

[150] Tobias Paxian and Bernd Becker. “Pacose: An Iterative SAT-based MaxSAT Solver”. In: *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*. 2020, p. 12 (cit. on p. 62).

[151] Joseph E. Reeves and Marijn J. H. Heule. “The impact of bounded variable elimination on solving pigeonhole formulas”. In: *Proceedings of Pragmatics of SAT* (2021) (cit. on p. 46).

[152] Julian Reisch, Peter Großmann, and Natalia Kliewer. “Stable Resolving-A Randomized Local Search Heuristic for MaxSAT”. In: *KI 2020: Advances in Artificial Intelligence: 43rd German Conference on AI, Bamberg, Germany, September 21–25, 2020, Proceedings 43*. Springer. 2020, pp. 163–175 (cit. on p. 64).

[153] Irina Rish and Rina Dechter. “Resolution versus search: Two strategies for SAT”. In: *Journal of Automated Reasoning* 24.1 (2000), pp. 225–275 (cit. on p. 47).

[154] John Alan Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *J. ACM* 12.1 (1965), pp. 23–41 (cit. on p. 27).

[155] Lawrence Ryan. “Efficient algorithms for clause-learning SAT solvers”. In: (2004) (cit. on p. 42).

[156] 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 (cit. on p. 62).

[157] Dale Schuurmans and Finnegan Southey. “Local search characteristics of incomplete SAT procedures”. In: *Artificial Intelligence* 132.2 (2001), pp. 121–150 (cit. on p. 45).

[158] Dale Schuurmans, Finnegan Southey, Robert C Holte, et al. “The exponentiated subgradient algorithm for heuristic boolean programming”. In: *IJCAI*. 2001, pp. 334–341 (cit. on p. 45).

[159] B Selman, H Levesque, and D Mitchell. “A new method for solving hard satisfiability problems, in proc. 10th National Conf. on Artificial Intelligence”. In: (1992) (cit. on p. 45).

[160] Bart Selman, Henry Kautz, et al. “Domain-independent extensions to GSAT: Solving large structured satisfiability problems”. In: *IJCAI*. Vol. 93. 1993, pp. 290–295 (cit. on p. 46).

[161] Bart Selman, Henry A Kautz, Bram Cohen, et al. “Local search strategies for satisfiability testing.” In: *Cliques, coloring, and satisfiability* 26 (1993), pp. 521–532 (cit. on pp. 64, 65).

[162] Bart Selman, Henry A Kautz, Bram Cohen, et al. “Noise strategies for improving local search”. In: *AAAI*. Vol. 94. 1994, pp. 337–343 (cit. on pp. 45, 46).

[163] Bart Selman, Henry A Kautz, Bram Cohen, et al. “Noise strategies for improving local search”. In: *AAAI*. Vol. 94. 1994, pp. 337–343 (cit. on p. 65).

[164] Claude E Shannon. “A symbolic analysis of relay and switching circuits”. In: *Electrical Engineering* 57.12 (1938), pp. 713–723 (cit. on p. 25).

[165] 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 (cit. on pp. 14, 35).

[166] Laurent Simon and Gilles Audemard. “Predicting learnt clauses quality in modern SAT solvers”. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI*. 2009, pp. 399–404 (cit. on pp. 25, 35, 38, 74).

[167] Raymond Smullyan. *First-Order Logic*. Second corrected. First published 1968 by Springer-Verlag. Dover Publications, New York, 1995 (cit. on pp. 98, 101).

[168] Sathiamoorthy Subbarayan and Dhiraj K Pradhan. “NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances”. In: *Theory and Applications of Satisfiability Testing: 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers 7*. Springer. 2005, pp. 276–291 (cit. on pp. 47, 48).

[169] Richard S Sutton. “Reinforcement learning: An introduction”. In: *A Bradford Book* (2018) (cit. on p. 41).

[170] Orestis Telelis and Panagiotis Stamatopoulos. “Heuristic backbone sampling for maximum satisfiability”. In: *Proceedings of the 2nd Hellenic Conference on Artificial Intelligence*. 2002, pp. 129–139 (cit. on p. 64).

[171] John Thornton, Stuart Bain, Abdul Sattar, et al. “A two level local search for MAX-SAT problems with hard and soft constraints”. In: *Australian Joint Conference on Artificial Intelligence*. Springer. 2002, pp. 603–614 (cit. on p. 64).

[172] John Thornton, Duc Nghia Pham, Stuart Bain, et al. “Additive versus multiplicative clause weighting for SAT”. In: *AAAI*. Vol. 4. 2004, pp. 191–196 (cit. on p. 45).

[173] Yiyuan Wang, Shaowei Cai, Jiejiang Chen, et al. “SCCWalk: An efficient local search algorithm and its improvements for maximum weight clique problem”. In: *Artificial Intelligence* 280 (2020), p. 103230 (cit. on p. 65).

[174] Zhao Xing and Weixiong Zhang. “MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability”. In: *Artificial intelligence* 164.1-2 (2005), pp. 47–80 (cit. on p. 56).

[175] Ramin Zabih and David A McAllester. “A Rearrangement Search Strategy for Determining Propositional Satisfiability.” In: *AAAI*. Vol. 88. 1988, pp. 155–160 (cit. on p. 55).

[176] Oleg Zaikin. “Inverting Step-Reduced SHA-1 and MD5 by Parameterized SAT Solvers”. In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2024 (cit. on p. 25).

[177] Lei Zhang and Fahiem Bacchus. “MAXSAT heuristics for cost optimal planning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 26. 1. 2012, pp. 1846–1852 (cit. on p. 53).

[178] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, et al. “Efficient conflict driven learning in a boolean satisfiability solver”. In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*. IEEE. 2001, pp. 279–285 (cit. on p. 35).

[179] Weixiong Zhang, Ananda Rangan, Moshe Looks, et al. “Backbone guided local search for maximum satisfiability”. In: *IJCAI*. Vol. 3. 2003, pp. 1179–1186 (cit. on p. 64).

[180] Wenjie Zhang, Zeyu Sun, Qihao Zhu, et al. “Nlocalsat: Boosting local search with solution prediction”. In: *arXiv preprint arXiv:2001.09398* (2020) (cit. on p. 44).

[181] Xindi Zhang, Shaowei Cai, and Zhihan Chen. “Improving cdcl via local search”. In: *SAT COMPETITION 2021* (2021), p. 42 (cit. on p. 79).

[182] Jiongzhi Zheng, Kun He, Zhuo Chen, et al. “Combining Hybrid Walking Strategy with Kissat MAB, CaDiCaL, and LStech-Maple”. In: *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*. University of Helsinki, 2022, pp. 20–21 (cit. on p. 73).

[183] Jiongzhi Zheng, Kun He, Jianrong Zhou, et al. “BandMaxSAT: A Local Search MaxSAT Solver with Multi-armed Bandit”. In: *Proceedings of IJCAI 2022*. 2022, pp. 1901–1907 (cit. on pp. 64–66).

[184] Neng-Fa Zhou. “Encoding the Hamiltonian Cycle Problem into SAT Based on Vertex Elimination (Short Paper)”. In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2024 (cit. on p. 25).