MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL
Number 38

Institut d'Investigació
en Intel·ligència Artificial

Consell Superior
d'Investigacions Científiques

Design and Implementation of Exact MAX-SAT Solvers

Jordi Planes

Foreword by Chu Min Li and Felip Manyà

2008 Consell Superior d'Investigacions Científiques
Institut d'Investigació en Intel·ligència Artificial
Bellaterra, Catalonia, Spain.

Series Editor
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Foreword by
Chu Min Li
Modélisation, Information et Systèmes
Université de Picardie Jules Verne

Felip Manyà
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Jordi Planes
Departament d'Informàtica i Enginyeria Industrial
Universitat de Lleida

Institut d'Investigació
en Intel·ligència Artificial

Consell Superior
d'Investigacions Científiques

NIPO: 472-08-052-7
ISBN: 978-84-00-08709-8
Dip. Legal: B-46479-2008

Ordering Information: Text orders should be addressed to the Library of the IIIA, Institut d'Investigació en Intel·ligència Artificial, Campus de la Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain.

A l'Àngels.

Contents

List of Figures

List of Tables

List of Algorithms

Foreword

A growing number of academic and real-world combinatorial problems are successfully being tackled by reducing them to Boolean Satisfiability (SAT), or to some well-known extensions of SAT such as Maximum Satisfiability (MAX-SAT), Multiple-Valued Satisfiability (Mv-SAT), Pseudo-Boolean Optimization (PBO), Quantified Boolean Formulas (QBF), and Satisfiability Modulo Theories (SMT). In fact, for many combinatorial search and reasoning tasks, the translation into a clausal formalism followed by the use of a satisfiability solver is often more effective than the use of a solver dealing with the original problem formulation.

This monograph, which is based on the Ph.D. dissertation of Dr. Jordi Planes, is concerned with the design, implementation and evaluation of exact MAX-SAT solvers, with special emphasis on devising good quality lower bounds, powerful inference techniques, clever variable selection heuristics, and suitable data structures. As a result of that research, four new MAX-SAT solvers —implementing the branch and bound scheme and incorporating the novel techniques presented here— are currently publicly available, and often used by the research community.

Among all the contributions, we would like to highlight the new inference rules, which transform a MAX-SAT instance into an equivalent and simpler MAX-SAT instance; and the computation of underestimations in the lower bound based on detecting disjoint unsatisfiable subformulas by applying unit propagation and failed literal detection. Such techniques are, nowadays, part of some of the most successful MAX-SAT solvers, and have shown, in the international evaluations of MAX-SAT solvers held so far, that they have a great impact on the solvers performance.

Finally, we hope that you enjoy reading this monograph, which is the fruit of the enthusiasm and effort that the author put into this scientific adventure which we had the pleasure to supervise.

Bellaterra, October 2008

Chu-Min Li
Université de Picardie Jules Verne

Felip Manyà
IIIA-CSIC

Abstract

The Propositional Satisfiability Problem (SAT) is the problem of determining whether a truth assignment satisfies a CNF formula. Nowadays, many hard combinatorial problems such as practical verification problems in hardware and software can be solved efficiently by encoding them into SAT.

In this thesis, we focus on the Maximum Satisfiability Problem (MAX-SAT), an optimization version of SAT which consists of finding a truth assignment that satisfies the maximum number of clauses in a CNF formula. We also consider a variant of MAX-SAT, called weighted MAX-SAT, in which every clause is associated with a weight and the problem consists of finding a truth assignment in which the sum of weights of violated clauses is minimum. While SAT is NP-complete and well-suited for encoding and solving decision problems, MAX-SAT and weighted MAX-SAT are NP-hard and well-suited for encoding and solving optimization problems.

This thesis is concerned with the design, implementation and evaluation of exact MAX-SAT solvers based on the branch and bound scheme, with special emphasis on defining good quality lower bounds, powerful inference techniques, clever variable selection heuristics and suitable data structures.

First, we have defined three original lower bound computation methods: star rule, UP, and UP enhanced with failed literal detection. All of them compute an underestimation of the number of clauses that will become unsatisfied if a partial assignment is completed. Such an underestimation is the number of disjoint subsets that can be declared unsatisfiable by deriving, in polynomial time, a resolution refutation from the clauses in the subset. The star rule considers subsets formed by n unit clauses and an n-ary clause which is the disjunction of the complementary literals of the literals occurring in the unit clauses; a linear unit refutation can be derived from those clauses. UP detects contradictions via unit propagation and identifies, for every contradiction, a subset of the clauses involved in the unit propagation from which a unit refutation can be derived. UP enhanced with failed literal detection allows to identify subsets from which both unit and non-unit refutations can be derived.

Second, we have defined a set of novel inference rules that transform a MAX-SAT instance ϕ into another MAX-SAT instance ϕ' in such a way that the number of unsatisfied clauses in ϕ is the same as the number of unsatisfied clauses in ϕ' for every assignment. All of them can be seen as unit resolution

refinements adapted to MAX-SAT.

Third, we have incorporated the lower bounds and the inference rules into a branch and bound algorithm in such a way that the computation of the lower bounds and the application of the inference rules is done simultaneously by inspecting the implication graph. As a result, we have developed a MAX-SAT solver, called MaxSatz, which was the best performing solver in the First MAX-SAT Evaluation, which was a co-located event of SAT-2006.

Fourth, we have defined extremely efficient lazy data structures for branch and bound MAX-SAT solvers with a static variable ordering. We have incorporated those data structures into a weighted MAX-SAT solver, called Lazy, which implements a lower bound and simple weighted MAX-SAT inference rules.

Finally, we have conducted a comprehensive experimental investigation that provides empirical evidence of the good performance profile of the lower bounds, inference rules, variable selection heuristics and data structures introduced. The results for both randomly generated and realistic problems show that the solvers developed in this thesis outperform state-of-the-art MAX-SAT and weighted MAX-SAT solvers on a wide range of instances.

Acknowledgments

First, I would like to thank my supervisors Dr. Chu Min Li and Dr. Felip Manyà. They have been continuously fostering enthusiasm and come along with new challenges. I feel extremely lucky working with them because I consider they are together the perfect Ph.D. supervisor: they have some complementary skills. Yet, they share something in common: generosity. I am in debt with them.

I also thank the members of the examining committee: Dr. Javier Larrosa, Dr. Daniel Le Berre, Dr. Jordi Levy, Dr. Hans van Maaren, and Dr. Pedro Meseguer. They have given valuable comments to improve the dissertation.

I would also like to thank the members of *Escola Politècnica Superior* with whom I shared coffees and friendship: Josep, Paula and Carlos, Tere, Carlos, Carles, Ramón, Cèsar, Fernando, Josep Lluís, Maite, ... Many thanks to *mes collègues amiénois* too, with whom I relished many more coffees and friendship in Le Cyrano: Stéphane, Sylvain, Sidney, Gilles and Laure.

Thanks to the *Universitat de Lleida* for their technical support and assistance. Specially to Carles, who maintains the cluster, and is relished that I have finished my Ph.D. dissertation (and the corresponding experimentation).

And finally, thanks to my wife Àngels. This thesis is dedicated to her, without whose love and support this would not have been possible.

Chapter 1

Introduction

As soon as an Analytical Engine exists,
it will necessarily guide the future course of the science.
Whenever any result is sought by its aid,
the question will then arise —
But what course of calculation can these
results be arrived at by the machine
in the shortest time?
Charles Babbage (1864)

1.1 Context

Since Charles Babbage, many computer scientists have been asking themselves the same question: *may I find a way to make the computer solve a problem in shorter time*? In the process of pursuing such a goal, several important computational problems have been identified as core problems in Computer Science. Among them it stands out the satisfiability problem (SAT), which is the problem of deciding if there exists a truth assignment that satisfies a propositional formula in conjunctive normal form (CNF formula). Such a problem is classified among combinatorial problems, which commonly imply finding values to a set of variables which are restricted by a set of constraints; in some cases the aim is to find a solution satisfying all the constraints (satisfaction problems), in other cases the aim is to find a solution satisfying as many constraints as possible (optimization problems).

In this thesis, we focus on an optimization version of SAT: the Maximum Satisfiability problem (MAX-SAT). This problem consists of finding a truth assignment that satisfies the maximum number of clauses in a CNF formula. We will see in the sequel that the MAX-SAT algorithms actually solve the equivalent problem of finding a truth assignment that falsifies the minimum number of clauses in a CNF formula, since this carries implementation benefits. Sometimes, we also consider a variant of MAX-SAT, called weighted MAX-SAT. In

weighted MAX-SAT, every clause has a weight and the problem consists of finding a truth assignment in which the sum of weights of violated clauses is minimal. While SAT is NP-complete [Coo71], both MAX-SAT and weighted MAX-SAT are NP-hard [GJ79].

We started our research on MAX-SAT in 2002, when SAT was —as it is nowadays— a central topic in Artificial Intelligence and Theoretical Computer Science. At that time, there were publicly available complete solvers such as Chaff [MMZ+01], GRASP [MSS99], Posit [Fre95], Relsat [BS97], and Satz [LA97a, LA97b], as well as local search solvers such as GSAT and WalkSAT [SK93, SKC94, SLM92]. There was also enough empirical evidence about the merits of the generic problem solving approach which consists of modeling NP-complete decision problems as SAT instances, solving the resulting encodings with a state-of-the-art SAT solver, and mapping the solution back into the original problem. This generic problem solving approach was competitive in a variety of domains, including hardware verification [MSG99, MMZ+01, VB03], quasigroup completion [AGKS00, KRA+01], planning [KS96, Kau06], and scheduling [BM00].

Despite the remarkable activity on SAT, there was a reduced number of papers dealing with the design and implementation of exact MAX-SAT solvers, and solving NP-hard problems by reducing them to MAX-SAT was not considered a suitable alternative for solving optimization problems. This is in contrast with what happened in the Constraint Programming community, where the Weighted Constraint Satisfaction Problem (Weighted CSP) was a problem attracting the interest of that community, which published a considerable amount of results about weighted CSP [MRS06].

We decided to start our research by designing and implementing branch and bound MAX-SAT solvers in the style of the exact solvers developed by Wallace and Freuder [WF96], and Borchers and Furman [BF99], which can be seen as an adaptation to MAX-SAT of the Davis-Logemann-Loveland (DLL) procedure [DLL62]. We thought that we could produce good performing MAX-SAT solvers by adapting to MAX-SAT the technology incorporated into the existing DLL-style SAT solvers, which were equipped with optimized data structures, clever variable selection heuristics, clause learning, non-chronological backtracking, randomization and restarts.

Before going into more technical details, let us introduce how works a basic branch and bound (BnB) algorithm for MAX-SAT: BnB explores the search space induced by all the possible truth assignments in a depth-first manner. At each node of the search tree, BnB compares the number of clauses falsified by the best complete assignment found so far —called *Upper Bound* (UB)— with the *Lower Bound* (LB), which is the number of clauses falsified by the current partial assignment plus an *underestimation* of the number of clauses that would become unsatisfied if the current partial assignment is extended to a complete assignment. Obviously, if $UB \leq LB$, a better assignment cannot be found from this point in the search, and BnB prunes the subtree below the current node and backtracks to a higher level in the search tree. If $UB > LB$, the current partial

assignment is extended by instantiating one more variable; which leads to create two branches from the current branch: the left branch corresponds to instantiate the new variable to false, and the right branch corresponds to instantiate the new variable to true. The solution to MAX-SAT is the value that UB takes after exploring the entire search tree.

At first sight, we observe two differences between BnB and DLL: unit propagation is not applied since it is unsound for MAX-SAT, and a lower bound has to be updated at each node of the search tree. On the one hand, the fact that unit propagation does not preserve the number of unsatisfied clauses led us to study new forms of inference for MAX-SAT. On the other hand, the fact of having to compute a lower bound at each node of the search tree led us to improve the existing lower bound computation methods.

Unit propagation is unsound for MAX-SAT, in the sense that the number of clauses falsified by an assignment is not preserved between a CNF formula ϕ and the formula obtained after applying unit propagation to ϕ. For example, if we apply unit propagation to $\phi = \{p_1, \neg p_1 \vee \neg p_2, \neg p_1 \vee p_2, \neg p_1 \vee \neg p_3, \neg p_1 \vee p_3\}$, we get two unsatisfied clauses. While assigning all the variables to false falsifies exactly one clause of ϕ. Therefore, if unit propagation is applied in BnB, non-optimal solutions can be obtained.

We devoted a part of this thesis to define sound and efficiently applied resolution rules for MAX-SAT that are, in a sense, the MAX-SAT counterpart of unit resolution. Let us see an example of inference rule: Given a MAX-SAT instance ϕ that contains three clauses of the form $l_1, l_2, \bar{l}_1 \vee \bar{l}_2$, where l_1, l_2 are literals, replace ϕ with the CNF formula

$$\phi' = (\phi \setminus \{l_1, l_2, \bar{l}_1 \vee \bar{l}_2\}) \cup \{\square, l_1 \vee l_2\}.$$

Note that the rule detects a contradiction from $l_1, l_2, \bar{l}_1 \vee \bar{l}_2$ and, therefore, replaces these clauses with an empty clause $\square$. In addition, the rule adds the clause $l_1 \vee l_2$ to ensure the equivalence between ϕ and ϕ'. An assignment that falsifies l_1 and l_2 then falsifies 2 of those 3 clauses, while any other asignment falsifies exactly 1 clause. The last clause added by the rule captures such a state.

The inference rules contribute by deriving new empty clauses, but it is also important to improve the underestimation of the number of clauses that will become unsatisfied if the current partial assignment is completed. Basically, when we started our research, the underestimations defined for MAX-SAT were based on counting the number of complementary unit clauses in the CNF formula under consideration.

We realized that a suitable underestimation is provided by the number of disjoint unsatisfiable subformulas which can be computed with an efficient procedure. In a first step [LMP05], we defined a powerful lower bound computation method based on detecting disjoint unsatisfiable subformulas by applying unit propagation. Once a contradiction is detected by unit propagation, we derive a unit resolution refutation, and the clauses involved in that refutation are taken as an unsatisfiable subformula. We repeat that process until we are not able to detect further unsatisfiable subformulas. We showed that this method, which

can be applied in time linear in the length of the CNF formula ($\mathcal{O}(ub \times |\phi|)$, where ub is an upper bound of the minimum number of unsatisfied clauses in $|\phi|$, and ϕ is the length of ϕ, leads to lower bounds of good quality. In a second step [LMP06], we enhanced our method by also applying failed literal detection. This way, we can detect resolution refutations (not necessarily unit resolution refutations) which can be computed efficiently and are beyond the reach of unit propagation.

Interestingly, we showed that inference techniques used in SAT cannot be applied to MAX-SAT because they can produce non-optimal solutions, but we applied such techniques to dramatically improve the computation of underestimations for lower bounds.

Besides defining original inference rules and good quality lower bounds, a constant concern of our work was to pay special attention to implementation issues and, in particular, to the definition of suitable data structures that allow one to perform as efficiently as possible the more common operations. Therefore, we defined optimized data structures for implementing the application of the inference rules and lower bounds, and investigated the use of very basic lazy data structures for implementing a simple and fast weighted MAX-SAT solver.

In the course of the thesis, we implemented four MAX-SAT solvers: AMP, Lazy, UP, and MaxSatz. The more sophisticated solver is MaxSatz, which incorporates the main contributions of our research. Further details about these solvers and the ideas behind can be found in the remaining chapters. At this point, we just would like to mention that MaxSatz was the best performing solver on all the sets of MAX-SAT instances that were solved in the *First MAX-SAT Evaluation*, a co-located event of the Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT-2006).

1.2 Objectives

The general objective of our research is the design, implementation and evaluation of exact algorithms for MAX-SAT with the ultimate goal of improving, and converting into a suitable alternative for solving optimization problems, the generic problem solving approach consisting of modeling optimization problems as MAX-SAT instances, solving the resulting encodings with a MAX-SAT solver, and mapping the solution back to the original problem.

The particular objectives of the thesis can be summarized as follows:

- Define underestimations of good quality for lower bound computation methods that allow one to prune as soon as possible the parts of the search space that do not contain any optimal solution.

- Define sound resolution-style inference rules for MAX-SAT whose application allows one to derive empty clauses as early as possible during the exploration of the search space.

- Design algorithms that combine the computation of underestimations and the application of inference rules in such a way that both operations can be done efficiently and accelerate the search for an optimal solution.

- Design and implement MAX-SAT solvers, equipped with optimized data structures and clever variable selection heuristics, that incorporate the MAX-SAT solving techniques we defined.

- Conduct an empirical evaluation of the techniques developed in this thesis, and in particular of the underestimations and inference rules devised. Identifying their strengths and weaknesses should allow us to gain new insights for developing more powerful MAX-SAT solving techniques.

- Conduct an empirical comparison between our solvers and the best performing state-of-the-art MAX-SAT solvers. Knowing the performance profile of our *competitors* can help improve the performance of our solvers.

- Identify MAX-SAT techniques that can be naturally extended to weighted MAX-SAT, and analyze the implementation issues that should be reconsidered to develop fast weighted MAX-SAT solvers.

1.3 Contributions

The main contributions of this thesis can be summarized as follows:

- We defined lower bound computation methods that detect disjoint unsatisfiable subformulas by applying unit propagation, and subsume most of the existing MAX-SAT lower bounds. As a result, we defined three new lower bounds: UP, UP^S, and UP^*. The main difference among them is the data structures that implement for storing unit clauses. These data structures have an impact on the number of unit clauses of the MAX-SAT instance under consideration that are used in the derivation of unit resolution refutations.

- We enhanced the previous lower bounds with failed literal detection, giving rise to three additional lower bounds: UP_{FL}, UP^S_{FL}, and UP^*_{FL}.

- We defined a set of original inference rules for MAX-SAT which are sound and can be applied efficiently. All of them can be seen as unit resolution refinements adapted to MAX-SAT.

- We have analyzed the time complexity of the lower bounds and the inference rules proposed. Most of them can be applied in time linear in the size of the CNF formula. In particular, we defined an algorithm that combines the application of powerful lower bounds and inference rules in time $\mathcal{O}(ub \times |\phi|)$, where ub is an upper bound of the minimum number of unsatisfied clauses and $|\phi|$ is the size of the CNF formula ϕ.

- We have extended some of the previous lower bounds and inference rules to weighted MAX-SAT.
- We designed and implemented four MAX-SAT solvers:

  AMP : This was our first solver. It extends the solver of Borchers and Furman [BF99] by incorporating a different variable selection heuristic and a more powerful underestimation.

  Lazy : This was our second solver. It deals with very simple lazy data structures that allow to perform quickly the most common operations. It is based on a static variable selection heuristic and solves both MAX-SAT and weighted MAX-SAT instances.

  UP : This was our third solver. The main implementation issues were adaptations to MAX-SAT of the technology implemented in the SAT solver Satz. The most powerful component of UP is the detection of disjoint unsatisfiable subformulas by unit propagation.

  MaxSatz : This was our last solver. It applies both the underestimations and the inference rules defined in this thesis.
- We conducted an empirical evaluation of the underestimations and inference rules developed in the thesis. We observed that UP^*_{FL} is usually the best performing underestimation for the testbed used and, in combination with our inference rules, gives rise to the best performance profile.
- We conducted an empirical comparison between our solvers and the best performing state-of-the-art MAX-SAT solvers. We observed that MaxSatz is extremely competitive and outperforms the rest of MAX-SAT solvers up to several orders of magnitude on a significant number of instances.

1.4 Publications

Some of the results presented in this thesis have already been published in journals and conference proceedings. The articles are chronologically listed and classified according to the solver used on it:

AMP

- Teresa Alsinet, Felip Manyà, and Jordi Planes. Improved branch and bound algorithms for Max-2-SAT and weighted Max-2-SAT. In *Proceedings of the 6th Catalan Conference on Artificial Intelligence (CCIA 2003)*, volume 100 of *Frontiers in Artificial Intelligence and Applications*, pages 435–442, P. Mallorca, Spain, 2003. IOS Press.
- Teresa Alsinet, Felip Manyà, and Jordi Planes. Improved branch and bound algorithms for Max-SAT. In *Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing (SAT 2003)*, pages 408–415, Portofino, Italy, 2003.

– Jordi Planes. Improved branch and bound algorithms for Max-2-SAT and weighted Max-2-SAT. In Francesca Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of LNCS, page 991, Kinsale, Ireland, 2003. Springer.

Lazy

– Teresa Alsinet, Felip Manyà, and Jordi Planes. A Max-SAT solver with lazy data structures. In *Proceedings of the 9th Ibero-American Conference on Artificial Intelligence (IBERAMIA 2004)*, volume 3315 of LNAI, pages 334–342, Puebla, Mexico, 2004. Springer.

– Teresa Alsinet, Felip Manyà, and Jordi Planes. A Max-SAT solver with lazy data structures. In Le Thi Hoai An and Pham Dinh Tao, editors, *Modeling, Computation and Optimization in Information Systems and Management Sciences (MCO 2004)*, pages 491–498, Metz, France, 2004. Hermes publishing.

– Teresa Alsinet, Felip Manyà, and Jordi Planes. Improved exact solver for weighted Max-SAT. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of LNCS, pages 371–377, St. Andrews, Scotland, 2005. Springer.

UP

– Chu Min Li, Felip Manyà, and Jordi Planes. Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In Peter van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005)*, volume 3609 of LNCS, pages 403–414, Sitges, Spain, 2005. Springer.

MaxSatz

– Chu Min Li, Felip Manyà, and Jordi Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, pages 86–91, Boston/MA, USA, 2006. AAAI Press.

After defending the doctoral dissertation, we published two journal articles containing its main contributions:

– Chu Min Li, Felip Manyà, and Jordi Planes. New Inference Rules for Max-SAT. In *Journal of Artificial Intelligence Research*, volume 30, pages 321–359, 2007.

– Teresa Alsinet, Felip Manyà, and Jordi Planes. An Efficient Solver for Weighted Max-SAT. In *Journal of Global Optimization*, volume 41, pages 61–73, 2008.

1.5 Overview

This section provides an overview of the thesis. We briefly describe the contents of each of the remaining chapters:

Chapter 2: Algorithms for SAT and MAX-SAT. In this chapter we provide an overview of techniques used in SAT and MAX-SAT solving. First, some basic concepts commonly used in satisfiability solving are introduced. Second, different techniques for SAT solving such as the DP algorithm and the DLL algorithm, recent efficient techniques in complete SAT solving, as well as representative local search algorithms are described. Third, the branch and bound algorithm to solve MAX-SAT is also introduced, with special attention to the techniques in lower bounds, inference rules and variables selection heuristics for MAX-SAT.

Chapter 3: Lower bounds. In this chapter we focus on computing lower bounds, the forecasting techniques for MAX-SAT. First, we review some state-of-the-art lower bounds. Second, we introduce the star rule, which is our first original lower bound computation method. Third, we define three original lower bounds that detect disjoint inconsistent subformulas by applying unit propagation. Fourth, we improve the previous lower bounds by adding failed literal detection. Finally, we report on the empirical evaluation of our lower bound computation methods.

Chapter 4: Inference rules. In this chapter we define a set of unit resolution refinements for MAX-SAT, describe an efficient way of implementing the application of the rules at each node of the search tree, and report on an experimental evaluation that provides empirical evidence that our rules can speed up a MAX-SAT solver several orders of magnitude.

Chapter 5: Implementing a weighted MAX-SAT solver. In this chapter we define a lower bound and a set of inference rules for weighted MAX-SAT, that are extensions of the MAX-SAT ones. We describe their implementation in an algorithm with a static variable selection heuristic and lazy data structures, and report on an experimental evaluation that provides empirical evidence of the good performance of the rules.

Chapter 6: Empirical comparison of MAX-SAT and weighted MAX-SAT solvers. In this chapter we first describe the best performing state-of-the-art MAX-SAT solvers, and the solvers we have designed and implemented in this thesis. Then, we report on an experimental comparison that provides empirical evidence that our solvers outperform the rest of the solvers in most of the solved instances.

Chapter 7: Conclusions. We briefly summarize the main contributions of the thesis, and point out some open problems and future research directions that we plan to tackle in the near future.

Finally, this thesis has an appendix, with enhanced inference rules that are provided as future work.

Chapter 2

Algorithms for SAT and MAX-SAT

This chapter provides some background information with the aim of making the thesis as self-contained as possible. Section 2.1 defines the syntax and semantics of CNF formulas, as well as SAT, MAX-SAT and weighted MAX-SAT. Section 2.2 describes the complete SAT algorithms Resolution, DP and DLL, two of the most representative SAT local search algorithms, GSAT and WalkSAT, and an overview of recent techniques in SAT algorithms. Section 2.3 introduces a basic MAX-SAT branch and bound algorithm, stressing its more important points; a brief of local search and approximation algorithms for MAX-SAT; and an overview of branch and bound algorithms for MAX-SAT, describing relevant lower bounds, inference rules and variable selection heuristics.

2.1 Definitions

Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a set of n propositional variables. A *literal* ℓ is a variable p_i or its negation $\neg p_i$. The complement of a literal ℓ, denoted by $\bar{\ell}$, is p if $\ell = \neg p$, and $\neg p$ if $\ell = p$. A *clause* is a disjunction of literals, and a *formula* in Conjunctive Normal Form (CNF formula) is a collection of clauses. In the satisfiability problem (SAT), a CNF formula is considered as a set of clauses. In the maximum satisfiability problem (MAX-SAT), a CNF formula is considered as a multiset of clauses. A clause with one literal is called *unit*, with two literals is called *binary*, and with three literals is called *ternary*. The size of a clause is the number of literals occurring in the clause, and the size of a CNF formula ϕ, denoted by $|\phi|$, is the sum of the sizes of their clauses.

A truth assignment is a mapping that assigns either the truth value 1 (true/T) or the truth value 0 (false/F) to each propositional variable. A truth assignment satisfies a literal p if p takes the value 1 (true/T), and satisfies a literal $\neg p$ if p takes the value 0 (false/F); satisfies a clause if it satisfies at least one literal of the clause; and satisfies a CNF formula if it satisfies all its clauses. A CNF

formula is satisfiable if there exists an assignment that satisfies the formula, otherwise it is unsatisfiable. A *tautology* is a CNF formula that is satisfied by any truth assignment. A clause with no literals is an *empty clause* (also referred to as *conflict*) and is denoted by $\square$. An empty clause is unsatisfied by any truth assignment.

A truth assignment is *complete* if, and only if, all the variables have been assigned; otherwise, it is *partial*. A partial truth assignment also partitions the clauses of a CNF formula into three sets: *satisfied* clauses, the clauses that contain a satisfied literal; *unsatisfied* clauses, the clauses in which all the literals are unsatisfied, and *unresolved* clauses, the clauses that the partial assignment makes them not to be decided. The unassigned literals of a clause are referred to as its *free literals*. In a search context, an unresolved clause is said to be *unit* if the number of its free literals is one. Similarly, an unresolved clause with two free literals is said to be *binary*, and an unresolved clause with three free literals is said to be *ternary*.

Example 2.1 *Let us consider a CNF formula ϕ having three clauses c_1, c_2 and c_3:*

$$\begin{array}{lcl} c_1 & : & (p_1 \vee p_2) \\ c_2 & : & (p_2 \vee \neg p_3) \\ c_3 & : & (p_1 \vee p_2 \vee p_3) \end{array}$$

Suppose that the current truth assignment is $\mathcal{A} : \{p_1 = F, p_3 = F\}$. This implies having clauses c_1 and c_3 unresolved and clause c_2 satisfied. Observe that clauses c_1 and c_3 are also unit due to p_2 being the only free literal. Hence, the CNF formula is unresolved.

Suppose that this assignment is extended with $p_2 = F$; i.e., $\mathcal{A}' : \{p_1 = F, p_2 = F, p_3 = F\}$. Then, clauses c_1 and c_3 become unsatisfied. This means that the CNF formula is unsatisfied by $\mathcal{A}'$, $\mathcal{A}'(\phi) = F$. Also, suppose that in the subsequent search we have the assignment $\mathcal{A}'' : \{p_1 = T, p_3 = F\}$. Clearly, all the clauses get satisfied and, therefore, the CNF formula is satisfiable, $\mathcal{A}''(\phi) = T$.

The SAT problem for a CNF formula ϕ is the problem of deciding whether there exists a satisfying assignment for ϕ. The MAX-SAT problem for a CNF formula ϕ is the problem of finding a complete assignment that maximizes the number of satisfied clauses in ϕ, or equivalently that minimizes the number of unsatisfied clauses in ϕ (named MIN-UNSAT).[1] Both problems are equivalent: Given an assignment that satisfies the maximum number of clauses in ϕ, it actually falsifies the minimum number of clauses in ϕ.

Example 2.2 *Let us consider an unsatisfiable CNF formula ϕ having 3 clauses: $(p_1 \vee p_2), (p_1 \vee \neg p_2), \neg p_1$. One solution for the MAX-SAT problem is the assignment $\mathcal{A} : \{p_1 = T, p_2 = T\}$, which satisfies the maximum number of clauses (2 clauses) and falsifies the minimum number of clauses (1 clause).*

[1] Some authors [LAS05] name MIN-UNSAT to the problem of finding minimally unsatisfiable subformulas.

A weighted clause is a pair (c, ω) such that c is a clause and ω is a cost associated with the clause. We make the usual assumption of weights being natural numbers, then the pair (c, ω) is clearly equivalent to having ω copies of clause c in our multiset. A weighted CNF formula is a multiset of weighted clauses. The weighted MAX-SAT problem is the problem of finding an assignment that minimizes the sum of weights associated to unsatisfied clauses (or equivalently, that maximizes the sum of weights associated to satisfied clauses) in a weighted CNF formula.

In SAT, two formulas ϕ_1 and ϕ_2 are equivalent if they are satisfied by the same set of assignments. In MAX-SAT, two formulas ϕ_1 and ϕ_2 are equivalent if both have the same number of unsatisfied clauses for every assignment. In weighted MAX-SAT, two formulas ϕ_1 and ϕ_2 are equivalent if the sum of the weights of unsatisfied clauses coincides for every assignment.

2.2 SAT algorithms

We describe the most popular complete methods —resolution, DP and DLL— for solving SAT, as well as the the local search algorithms GSAT and WalkSAT. See [GPFW97, AM03, GKSS07] for surveys about SAT algorithms.

2.2.1 Resolution

Resolution [Rob65] is an inference rule that provides a complete inference system by refutation. Given two clauses c_1, c_2, called *parent clauses*, then r is a *resolvent* of c_1 and c_2 if there is one literal $\ell \in c_1$ such that $\bar{\ell} \in c_2$, and r has the form

$$r = (c_1 \setminus \{\ell\}) \cup (c_2 \setminus \{\bar{\ell}\}).$$

The resolution step for a CNF formula ϕ, denoted by $Res(\phi)$, is defined as

$$Res(\phi) = \phi \cup \{r \mid r \text{ is a resolvent of two clauses in } \phi\}$$

Resolution is the application of resolution steps to a formula ϕ until $Res(\phi) = \phi$; i.e., no more resolvents can be derived. Then, the formula is unsatisfiable if $\square \in \phi$; otherwise, ϕ is satisfiable. Algorithm 2.1 represents this procedure [Sch89].

2.2.2 The Davis-Putnam procedure

The first effective method for producing resolution refutations [Vel89] was the Davis-Putnam procedure (DP) [DP60]. The method iteratively simplifies the formula until the empty clause is generated or until the formula is empty. DP consists of three rules:

1. *Unit Propagation*, also referred to as Boolean constraint propagation [ZM88], is the iterated application of *Unit Clause* (UC) *rule* (also referred to as *one-literal rule*) until an empty clause is derived or there are no unit clauses left. If a clause is unit, then the sole free literal must be assigned value

Algorithm 2.1: `Resolution`(ϕ)

Output: Satisfiability of ϕ
Function `Resolution(` ϕ *: CNF formula*`)` **: Boolean**
  repeat
    $\phi' \leftarrow \phi$
    $\phi \leftarrow Res(\phi)$
  until $\square \in \phi \ \vee \ \phi = \phi'$
  if $\square \in \phi$ **then return** false
  else return true

true. Being $\{\ell\}$ a unit clause, UC consists of removing all clauses in ϕ with literal ℓ and removing all occurrences of literal $\bar{\ell}$.

2. *Pure literal rule* (also referred to as monotone literal rule). A literal is pure if its complement does not occur in the CNF formula. The satisfiability of a CNF formula is unaffected by satisfying those literals. Therefore, all clauses containing a pure literal can be removed.

3. *Resolution* is applied in order to iteratively eliminate each variable from the CNF formula. In order to do so, DP does not apply general resolution, but a refinement (a restriction) of the resolution method, known as *variable elimination*: Let $\mathcal{C}_\ell$ be the set of clauses containing ℓ and $\mathcal{C}_{\bar{\ell}}$ the set of clauses containing $\bar{\ell}$, the method consists of generating all the non-tautological resolvents using all clauses in $\mathcal{C}_\ell$ and all clauses in $\mathcal{C}_{\bar{\ell}}$, and then removing all clauses in $\mathcal{C}_\ell \cup \mathcal{C}_{\bar{\ell}}$. After this step, the CNF formula contains neither ℓ nor $\bar{\ell}$.

The pseudo-code for the Davis-Putnam procedure is given in Algorithm 2.2. The algorithm selects a variable to be eliminated among the shortest clauses. Each time a variable is eliminated, the number of clauses in the CNF formula may grow quadratically in the worst case. Therefore, the worst-case memory requirement for algorithm DP is exponential. In practice, DP can only handle SAT instances with tens of variables because of this memory explosion problem [Urq87, CS00]. The procedure stops applying resolution when the CNF formula is found to be either satisfiable or unsatisfiable. It is declared to be unsatisfiable whenever a conflict is reached. If no conflict is reached, the CNF formula is declared to be satisfiable.

Example 2.3 *Given the following CNF formula, we demonstrate its satisfiability using algorithm DP:*

$$(p_1), (p_1 \vee p_2), (p_2 \vee p_4), (\neg p_1 \vee p_3 \vee \neg p_4), (p_3 \vee p_5), (\neg p_1 \vee \neg p_3 \vee \neg p_5)$$

We show the steps applied by algorithm DP using a table. In the first column the input formula in CNF format is displayed, where each line represents a different clause. The rest of the columns represent the result of applying UC. The table

Algorithm 2.2: `DavisPutnam`(ϕ)

Output: Satisfiability of ϕ
Function `DavisPutnam`(ϕ : *CNF formula*) **: Boolean**
  `UnitPropagation`(ϕ)
  `PureLiteralRule`(ϕ)
  if $\phi = \varnothing$ **then return** true
  if $\square \in \phi$ **then return** false
  $\ell \leftarrow$ literal in $c \in \phi$ having c the minimum length
  $\mathcal{R}_\ell \leftarrow$ all possible non-tautological resolvent clauses between all clauses in $\mathcal{C}_\ell$ and all clauses in $\mathcal{C}_{\bar{\ell}}$
  return `DavisPutnam`($\phi \cup \mathcal{R}_\ell \setminus (\mathcal{C}_\ell \cup \mathcal{C}_{\bar{\ell}})$)

*below shows the application of the rule to literal p_1. Removed clauses are marked with a '$\times$' and modified clauses are displayed in **bold** face.*

ϕ	p_1
(p_1)	$\times$
$(p_1 \vee p_2)$	$\times$
$(p_2 \vee p_4)$	$(p_2 \vee p_4)$
$(\neg p_1 \vee p_3 \vee \neg p_4)$	$(\mathbf{p_3} \vee \mathbf{\neg p_4})$
$(p_3 \vee p_5)$	$(p_3 \vee p_5)$
$(\neg p_1 \vee \neg p_3 \vee \neg p_5)$	$(\mathbf{\neg p_3} \vee \mathbf{\neg p_5})$

In a second step, DP applies the pure literal rule. The table below shows the application of the rule to literals p_2 and $\neg p_4$.

ϕ'	p_2	$\neg p_4$
$(p_2 \vee p_4)$	$\times$	
$(p_3 \vee \neg p_4)$	$(p_3 \vee \neg p_4)$	$\times$
$(p_3 \vee p_5)$	$(p_3 \vee p_5)$	$(p_3 \vee p_5)$
$(\neg p_3 \vee \neg p_5)$	$(\neg p_3 \vee \neg p_5)$	$(\neg p_3 \vee \neg p_5)$

And finally, DP applies resolution. The table below shows the elimination of variable p_3 by resolution. Observe that a tautological clause appears, and is removed by the method.

ϕ''	p_3	
$(p_3 \vee p_5)$	$\times$	
$(\neg p_3 \vee \neg p_5)$	$(p_5 \vee \neg p_5)$	$\times$

At the end, the CNF formula becomes empty. Thus, the original CNF formula is satisfiable.

2.2.3 The Davis-Logemann-Loveland procedure

The vast majority of state-of-the-art complete SAT algorithms are built upon the backtrack search algorithm of Davis, Logemann and Loveland (DLL) [DLL62].

Algorithm 2.3: `DavisLogemannLoveland`(ϕ)

Output: Satisfiability of ϕ

Function `DavisLogemannLoveland`(ϕ : *CNF formula*) **: Boolean**

```
UnitPropagation(φ)
PureLiteralRule(φ)
if φ = ∅ then return true
if □ ∈ φ then return false
ℓ ← literal in c ∈ φ having c the minimum length
return (DavisLogemannLoveland(φ_ℓ) ∨
DavisLogemannLoveland(φ_ℓ̄))
```

DLL replaces the application of resolution in DP by the splitting of the CNF formula in two subproblems. The first subproblem $\phi_{\bar{\ell}}$ is the application of UC over ϕ with $\bar{\ell}$, and the second subproblem ϕ_ℓ is the application of UC over ϕ with ℓ. Then, ϕ is unsatisfiable if and only if ϕ_ℓ and $\phi_{\bar{\ell}}$ are unsatisfiable. This method is shown in Algorithm 2.3.

Procedure DLL essentially constructs a binary search tree in a depth-first manner (e.g., Figure 2.1), each leaf of the search tree represents a dead end where an empty clause is found, except eventually one for a satisfiable problem. Using a variable selection heuristic, the branching variables are selected to reach a dead end as early as possible; i.e., to minimize the length of the current path in the search tree.

Example 2.4 *The search tree for the CNF formula below is displayed in Figure 2.1.*

$$(p_1 \vee p_5), (p_1 \vee \neg p_6), (p_1 \vee \neg p_2 \vee p_4), (p_1 \vee p_2 \vee \neg p_4), (\neg p_2 \vee \neg p_4), (p_2 \vee p_4) \wedge (\neg p_1 \vee \neg p_2), (p_2 \vee p_3), (p_1 \vee p_2 \vee p_3)$$

Solid lines are for splitting assignments, and dashed lines for unit propagation and monotone literal assignments. Black nodes mark whenever a conflict is found.

The authors in [DLL62] identified three advantages of DLL over DP:

1. DP increases the number and length of the clauses rather quickly. DLL never increases the length of clauses. The worst-case space complexity is exponential in DP and polynomial in DLL.

2. Many duplicated clauses may appear after resolution in DP, and seldom after splitting in DLL.

3. DLL often can yield new unit clauses, while DP not often will.

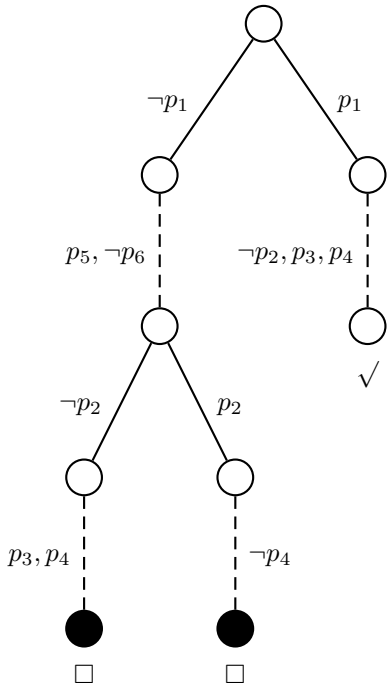


Figure 2.1: Search tree for DLL applied to Example 2.4.

2.2.4 Local search procedures for SAT

One of the weaknesses of complete methods (e.g., DP and DLL) is their inability to solve hard random 3-SAT instances with more than 700 propositional variables within a "reasonable" amount of time [CM97, DD01]. This drawback can be skipped using incomplete[2] local search methods like GSAT [SLM92] and WalkSAT [SKC94]. These procedures are able to solve hard instances with more than 100,000 variables, though completeness is lost.

The search is done by initializing the search at some point (a complete assignment) and from there to iteratively move from one search space position to a neighboring position. The decision on each step is based on information about the local neighborhood only.

Most local search SAT algorithms use a 1-flip neighborhood relation for which two truth value assignments are neighbors if they differ in the truth value of exactly one variable. Thus, the local search steps modify the truth value assigned

[2]An incomplete method in SAT can find a satisfying assignment, but cannot prove the unsatisfiability of a CNF formula. If a solution is found, the formula is declared satisfiable and the algorithm terminates successfully; but if the algorithm fails to find a solution, no conclusion can be drawn.

Algorithm 2.4: `LocalSearch`(ϕ) : Outline of a general local search procedure for SAT

```
Output: Satisfying assignment of φ or 'no solution found'
for 1 to maxTries do
    A ← initAssign(φ)
    for 1 to maxSteps do
        if A satisfies φ then return A
        else
            p ← chooseVariable(φ, A)
            A ← A with truth value of p flipped
return 'no solution found'
```

to one propositional variable; such a move is called a *variable flip*. The main difference between different local search algorithms for SAT is in the step function, that is, in the strategy used to select the variable to be flipped next.

Local search algorithms can get trapped in local minima and plateau regions of the search space, leading to premature stagnation of the search. One of the simplest mechanisms for avoiding premature stagnation of the search is *random restart*, which reinitializes the search if after a fixed number of steps no solution has been found. Random restart is used in almost every local search algorithm for SAT.

A general outline of a local search algorithm for SAT is given in Algorithm 2.4. The generic procedure initializes the search at some complete truth assignment (`initAssign`(ϕ)) and then iteratively selects a variable according to the input CNF formula (`chooseVariable`(ϕ,A)) and the current assignment, and flips this variable. If after a maximum of *maxSteps* flips no solution is found, the algorithm restarts from a new randomly generated initial assignment. If after a given number *maxTries* of such tries still no solution is found, the algorithm terminates unsuccessfully.

In the following, we focus on the GSAT and the WalkSAT algorithms, which have provided a major driving force in the development of local search algorithms for SAT [SHR01, HS04].

GSAT algorithm

The GSAT algorithm was introduced in 1992 [SLM92]. It is based on a rather simple idea: GSAT tries to maximize the number of satisfied clauses by a greedy ascent in the space of truth assignments. Variable selection in GSAT and most of its variants is based on the *score* of a variable p under the current assignment $\mathcal{A}$; which is defined as the difference between the number of clauses unsatisfied by the assignment obtained by flipping p in $\mathcal{A}$ and the number of clauses *unsatisfied* by $\mathcal{A}$.

The basic GSAT algorithm uses the following instantiation of the procedure *chooseVariable*(ϕ, A): In each local search step, one of the variables with maximal

score is flipped. If there are several variables with maximal score, one of them is randomly selected according to a uniform distribution.

WalkSAT algorithm

The WalkSAT algorithm was described by Selman, Kautz, and Cohen [SKC94] in 1994. It is based on a 2-stage variable selection process focused on the variables occurring in currently unsatisfied clauses. For each local search step, in a first stage, a currently unsatisfied clause c' is randomly selected. In a second stage, one of the variables appearing in c' is then flipped to obtain the new assignment.

Thus, while the GSAT algorithm is characterized by a static neighborhood relation between assignments with Hamming distance one, in WalkSAT the variable to be flipped is no longer picked among all variables but from a randomly selected unsatisfied clause [SHR01].

2.2.5 Overview of SAT algorithms

There have been developed many algorithms that have improved the above described algorithms. One of the main driving forces has been the SAT competition[3], organized by Le Berre and Simon since 2002 [BS03, SB04, SBH05, BS06]. In such a competition, the last advances in SAT solvers race each other, bringing winners every year in three benchmark categories: industrial, handmade and random.

In the following, we present an overview of SAT algorithms following three lines:

1. the algorithms that improved algorithm DLL with the implementation of better variable selection heuristics, learning techniques, efficient data structures, reasoning about special structures in SAT instances and restarts;

2. the algorithms based on local search methods; and

3. other algorithms.

Algorithms based on DLL

In this section we focus on important points to consider when designing and implementing SAT solvers: the variable selection heuristic, the data structures, clause learning, application of restarts and reasoning on special structures.

Variable selection heuristics The variable selection heuristic in algorithm DLL is decisive for finding as quick as possible a solution in SAT [MS99]. A bad heuristic can lead to explore the whole search tree, whereas a good heuristic allows to cut several branches, and even not to traverse more than a single branch in the best case.

[3]The results can be checked at `http://www.satcompetition.org/`.

The original variable selection heuristic in algorithm DP selects a variable of a literal among the shortest clauses. The variable selected is used to apply resolution and is selected after applying unit clause rule and pure literal rule. This choice is justified by the fact that the length of the resolvent clauses depends on the length of the parent clauses. For instance, in the case all the clauses have two literals, resolvents have at most two literals. Furthermore, this heuristic enforces the unit clause rule. The same heuristic is used in the original description of algorithm DLL to select the next branching variable.

Let ϕ be the following CNF formula:

$$\begin{aligned} \phi \quad : \quad & (\neg p_1 \vee p_2) \ , \ (\neg p_2 \vee p_4 \vee \neg p_3) \\ & (p_1 \vee \neg p_5) \ , \ (p_2 \vee p_4 \vee p_6) \\ & (\neg p_2 \vee p_4 \vee p_6) \ , \ (p_2 \vee \neg p_3 \vee p_4) \end{aligned} \tag{2.1}$$

The shortest clauses in ϕ are $(\neg p_1 \vee p_2)$ and $(p_1 \vee \neg p_5)$, hence DP heuristic chooses any of the variables p_1, p_2 or p_5.

The MOMS (Maximum Occurrences in clauses of Minimum Size)[DABC93, Pre93] heuristic is an improvement of the previous heuristic. It selects the variable having the maximum number of occurrences in clauses of minimum size. Intuitively, these variables allow to well exploit the power of unit propagation and to augment the chance to reach an empty clause [Fre95].

The shortest clauses in ϕ are $(\neg p_1 \vee p_2)$ and $(p_1 \vee \neg p_5)$, hence MOMS heuristic chooses variable p_1.

Two-sided Jeroslow-Wang (JW) heuristic [JW90, HV95] is based on the same principle as MOMS heuristic. It gives the possibility of being chosen to the variables that appear in the shortest clauses. In contrast with MOMS, the number of occurrences in the rest of clauses is also involved. The possibility that a variable is selected by JW is inversely proportional to the size of the clauses in which it appears. JW uses a function J that takes as input the literal ℓ and returns a weight for such a literal:

$$J(\ell) = \sum_{\{c \in \phi | \ell \in c\}} 2^{-|c|},$$

where $|c|$ is the number of literals in clause c. Heuristic JW chooses a variable p that maximizes $J(p) + (J\neg p)$. In this heuristic the weight given to a literal with an occurrence in a binary clause is equivalent to the occurrence in two ternary clauses[4].

[4]In [LA97a, XZ05] a different proportion is considered $J(\ell) = \sum 5^{-|c|}$, and is referred to as MOMS.

Let ϕ be the CNF formula 2.1. With function J, one can get: $J(p_1) = 0.25$, $J(\neg p_1) = 0.25$, $J(p_2) = 0.5$, $J(\neg p_2) = 0.25$, $J(p_3) = 0$, $J(\neg p_3) = 0.25$, $J(p_4) = 0.5$, $J(\neg p_4) = 0$, $J(p_5) = 0$, $J(\neg p_5) = 0.25$, $J(p_6) = 0.25$, $J(\neg p_6) = 0$. The variable p_2 with $J(p_2) + J(\neg p_2) = 0.75$ is chosen by heuristic JW.

Another set of heuristics are based on the application of unit propagation; e.g. POSIT [Fre95] and Tableau [CA96]. They have been proved useful and allow to exploit the power of unit propagation and the detection of failed literals. A *failed literal* is a literal whose addition to a CNF formula brings the empty clause after unit propagation. Given a variable p, a unit propagation heuristic examines p by respectively adding the unit clause p and $\neg p$ to a CNF formula, and independently makes two unit propagations. The real effect of the unit propagations is then used to weight p and detect failed literals. Thus, by considering $w(\ell)$ and $w(\bar{\ell})$, the number of clauses reduced respectively by ℓ and $\bar{\ell}$, this heuristics consists of choosing the literal which maximizes at the same time $w(\ell)$ and $w(\bar{\ell})$. If literal ℓ is a failed literal, then $\bar{\ell}$ is fixed. This approach makes possible to better prevent the consequences that the choice of the literal will produce.

Let ϕ be the CNF formula 2.1. With function w, one can get the following values: $w(p_1) = 3$, $w(\neg p_1) = 1$, $w(p_2) = 2$, $w(\neg p_2) = 4$, $w(p_3) = 2$, $w(\neg p_3) = 0$, $w(p_4) = 0$, $w(\neg p_4) = 4$, $w(p_5) = 4$, $w(\neg p_5) = 0$, $w(p_6) = 0$, $w(\neg p_6) = 2$. The variable p_2 with $w(p_2) = 2, w(\neg p_2) = 4$ is chosen by a unit propagation heuristic.

Since examining a variable by two unit propagations is time consuming, two major problems remain open: should one examine every free variable by unit propagation at every node of a search tree? Otherwise, what are the variables to be examined at a search tree node? In [LA97a], the authors try to experimentally address these two questions to obtain an optimal exploitation of the unit propagation heuristic. They define predicate $PROP_z$ at a search tree node whose meaning is the set of variables to be examined at that node. $PROP_z$ is defined as follows: if there are more than T (parameter empirically set to 10) variables occurring both negatively and positively in binary clauses and having at least 4 occurrences, then only such variables are examined using a unit propagation heuristic; otherwise, if there are more than T variables occurring both negatively and positively in binary clauses and having at least 3 occurrences, then only all these variables are examined using a unit propagation heuristic; otherwise, all free variables are examined.

An alternative approach, particularly good performing for random 3-SAT, is to select a variable that is likely to be a backbone variable [DD01, KSTW05]. A backbone literal is a literal that must be true in all the solutions to a given instance. Given a CNF formula ϕ, this heuristics tries on variables that belong (in fact, are expected to belong) to the backbone of ϕ. If backbone variables are selected first, the algorithm searches through less branches, speeding up the

solver. Heuristics based on unit propagation and backbone are usually effective on computationally difficult random SAT instances.

The previous heuristics were created without the addition of learning techniques into SAT solvers. Marques-Silva [MS99, LMS05] compared several DLL heuristics with a solver applying learning, GRASP [MSS96b], using real-world SAT instances, and found that none was a clear winner. The two following heuristics are thought for this kind of solvers, focusing on a kind of locality rather than formula simplification [Mit05], i.e. the heuristic has no information of the whole formula, but of the recent changes performed on it. For the solver Chaff [MMZ$^+$01, ZM02, Zha03], the authors proposed a branching heuristic called Variable State Independent Decaying Sum (VSIDS). This heuristic keeps a score for each literal. Initially, the scores are the number of occurrences of a literal in the initial problem instance. Because of the learning mechanism, clauses are added to the formula as the search progresses. VSIDS increases the score of a literal by a constant whenever an added clause contains the literal. The VSIDS score is a literal occurrence count with higher weight on the variables occurring in more recently added clauses. Periodically, it computes all literal scores as $s(l) = r(l) + s(l)/2$, where $s(l)$ is the score for literal l, and $r(l)$ is the number of occurrences of l in a conflict clause since the previous update. VSIDS will choose the free variable with the highest combined score to branch.

Goldberg and Novikov went a step forward with the heuristic used in BerkMin [GN01]. This heuristic responds more dynamically to recently learned clauses, with a new scoring computation $s(l) = r(l) + s(l)/4$, that penalizes the oldest variables. Scores are incremented for all variables used in the conflict clause derivation, not just those in the conflict clause. The BerkMin heuristic prefers to branch on variables involved in the derivation of the most recently learned clause.

Recent versions of Chaff (e.g. zChaff [MFM04]) use a heuristic that combines BerkMin heuristic with a version of heuristic VSIDS, and a scheme for deleting part of the current assignment when a newly derived conflict clause is very large, as a means of trying to keep conflict clause sizes small.

Efficient data structures. The performance of the DLL procedure critically depends upon the care taken of the implementation. SAT solvers spent much of its time in the unit propagation procedure [Zha97, LMS02], and there have been many attempts to improve the unit propagation implementation. A simple and intuitive approach consists in keeping counters for each clause. This scheme is attributed to Crawford and Auton [CA93] by [ZS96]. Similar schemes are subsequently employed in many solvers such as GRASP [MSS99], Relsat [BS97] and Satz [LA97a]. For example, in GRASP each clause keeps two counters, one for the satisfied literals in the clause and another for the unsatisfied literals in the clause. Each variable has two lists that contain all the clauses where that variable appears with positive and negative polarity. When a variable is assigned a value, all the clauses that contain this literal will have their counters updated. If a clause count of unsatisfied literals becomes equal to the total

number of literals in the clause, then it is a conflicting clause. If a clause count of unsatisfied literals is one less than the total number of literals in the clause and the count of satisfied literals is null, then the clause is a unit clause. A counter based unit propagation procedure is easy to understand and implement, but this scheme is not always the most efficient one.

As it is pointed out in [ZM02], Zhang and Stickel [ZS96], in order to speed up this procedure, created a new data structure in solver SATO: head/tail lists. In this mechanism, each clause has two pointers associated with it, called the head and tail pointer respectively. A clause stores all its literals in an array. Initially, the head pointer points to the first literal of the clause and the tail pointer points to the last literal of the clause. Each variable keeps four linked lists that contain pointer to clauses. Each of these lists contains the pointers to the clauses that have their head/tail literal in positive/negative polarity for a given variable. Whenever a variable is assigned, only two of the four lists will be examined. The head/tail list method is faster than the counter-based scheme because is more efficient when a variable is assigned. The main goal of this data structure is the detection of unit clauses, and is specially efficient in unit propagation. For both the counter-based algorithm and the head/tail list-based algorithm, undoing a variable assignment during backtrack has about the same computational complexity as assigning the variable.

In Chaff, the authors proposed another unit propagation method called 2-literal watching. Similar to the head/tail list method, 2-literal watching also has two special literals for each clause called watched literals. Each variable has two lists containing pointers to all the watched literals in either polarity. In contrast to the head/tail list scheme in SATO, there is no imposed order on the two pointers within a clause, and each of the pointers can move in either direction. The main advantage of this method is the fact that unassigning a variable can be done in constant time. This data structure was also used in solvers BerkMin and MiniSat [NE03].

Clause Learning. Many industrial SAT instances bear a pattern, that usually means that conflicts found throughout the search are due to sets of related clauses. Once one of this unsatisfiable sets is found, the reasons that caused the conflict can be stored adding redundant clauses. These clauses will facilitate to find unsatisfied clauses earlier in future branches [BS94, BGS99]. To find the reasons, a conflict graph is created (cf. Section 3.3.1), which helps to analyse and learn the reason of the failure. This technique is called *clause learning* [MSS96a, MSS96b], or conflict driven clause learning and is used in solvers like GRASP, Chaff, BerkMin, MiniSat and Siege [Rya04].

Restarts. Another problem with a complete procedure like DLL is that a bad decision in the branching heuristic can be very costly. Bad decisions made in the top of the search tree can lead to bad search branches, and therefore to a waste of time. Gent and Walsh [GW93a] identified SAT instances which were typically easy but could occasionally trip up the DLL procedure. Sometimes, this could

be explained by the heavy tailed behaviour [GSCK00]. Gomes, Selman and Kautz [GSK98, KHR+02] showed that a strategy of randomization and rapid restarts can often be effective attacking such early mistakes. The restart cutoff can be gradual [GSK98, BMS00] or fixed [GSCK00]. Restarting with increasing cutoff [Hua07] is used in solver MiniSat, and with fixed cutoff in solvers Chaff, BerkMin and Siege.

Reasoning on special structures in SAT instances. Given that many problems like pigeonhole or graph coloring involve a great deal of symmetry in their arguments, a variety of authors have suggested extending Boolean representation or inference in a way that allows this symmetry to be exploited directly. The basic idea is to add so-called symmetry-breaking clauses to the original formula, clauses that break the existing symmetry without affecting the overall satisfiability of the formula [CGLR96, ASM06, BS07]. Rather than modifying the set of clauses in the problem, it is also possible to modify the notion of inference, so that once a particular conflict has been derived, symmetric equivalents can be derived in a single step [Kri85].[5]

Another explored deduction mechanism for special structured instances is equivalence reasoning. Solver eqsatz [Li03] incorporates equivalence reasoning into the solver Satz, and its authors found that it is effective on some particular classes of benchmarks (e.g., Dubois[6]). In that work, the equivalence reasoning is accomplished by a pattern-matching scheme for equivalence clauses. In particular, finding equivalences of the type $p \leftrightarrow q$ can reduce the number of variables and clauses of the formula, since variables p and q can be collapsed into one variable. A related deduction mechanism was proposed in [LMS01]. There, the authors propose to include more patterns in the matching process for simplification purposes. A more complex equivalence reasoning, with several steps, is performed in [WvM98, HDvMvZ04] as a pre-processing.

Improved local search SAT algorithms

Following the steps of GSAT and WalkSAT, the most relevant local search algorithms developed in the last years are:

HSAT [GW93b] by Ian Gent and Toby Walsh. An improvement of GSAT, which flips the variable that was flipped longest ago.

TSAT [MSG97] GSAT algorithm with tabu search, by Bertrand Mazure, Lakhdar Saïs and Eric Grégoire.

novelty [MSK97] by McAllester, Selman, Kautz. This strategy sorts the variables by the total number of clauses that the variable falsifies, breaking ties in favor of the least recently flipped variable.

[5] Symmetry breaking has acquired such an interest in the research community that a new conference has been created, International Symmetry Conference (`http://isc.dcs.st-and.ac.uk/`), hold in Scotland in 2007.

[6] Benchmark available at `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/` or `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.

novelty+ [Hoo99] by Holger Hoos, novelty with random walk and a user fixed noise in the choice of flip. Another variant of the same author is *adapt novelty+*, with self-fixed (adaptive) noise in the choice of flip.

SAPS [HTH02] by Holger Hoos et al. A weight is given to each clause, incrementing the weight to unsatisfied clauses. The variable in more clauses of maximum weight is chosen.

g2wsat [LH05b] by Li and Huang, and *adaptg2wsat*, which chooses a promising decreasing variable. A variable is decreasing if flipping it would decrease the number of unsatisfied clauses. A variable is promising decreasing if it becomes a decreasing variable after flipping another variable.

VW by Steve Prestwitch, in SAT 2005 Competition. A weight is given to each variable, which is increased in variables that are often flipped. The variable with minimum weight is chosen.

Most of these techniques can be checked in solver UBCSAT [TH04], from the University of British Columbia.

Recently, novel local search algorithms for SAT have been defined: (i) a complete local search method for SAT [FR04] wherein each step is a resolution step instead of a variable assignment; and (ii) two local search algorithms for unsatisfiability [PL06], one with a search space of proof graphs, and a second one with resolvent multisets.

Other incomplete algorithms

Finally, we mention survey propagation, a method inspired in results from Statistical Physics [BMZ05]. This is not a local search method, but it is incomplete. It has demonstrated to be a very efficient approach, in the case of random k-SAT, mainly in the threshold point.

2.3 MAX-SAT algorithms

We present the most representative MAX-SAT algorithms that have been developed to solve the problem both in an exact manner and with local search. The exact algorithms are improvements of the branch and bound scheme with better forecasting of bad branches (i.e., lower bounds), efficient transformations of the formula into an easier one (i.e., inference rules), and good orderings of the variables (variable selection heuristics). Local search algorithms are adaptations of SAT local search algorithms. At the end, we present the solvers submitted to the MAX-SAT evaluation, which is expected to be one of the driving forces in the development of efficient MAX-SAT solvers, as the SAT competition has been for SAT solvers.

Algorithm 2.5: `MaxSatBnB`(ϕ) : Branch and Bound for MAX-SAT

Output: The minimum number of clauses of CNF formula ϕ that can be unsatisfied by an assignment

```
Function MaxSatBnB (φ : CNF formula) : Natural
    InferenceRules(φ)
    if φ = ∅ or φ only contains empty clauses then
        return EmptyClauses(φ)
    if LowerBound(φ) ≥ UpperBound(φ) then
        return UpperBound(φ)
    p ← SelectVariable(φ)
    return min( MaxSatBnB(φ_¬p), MaxSatBnB(φ_p))
```

2.3.1 Branch and Bound

In MAX-SAT, once a solution is found, the search cannot be stopped like in SAT, because MAX-SAT is not an NP problem but an NP-hard problem.[7] This makes the algorithm to explore all possible branches in the pursue of an optimal solution. Through the searching, the best solution found so far is stored in order to compare it with the solutions to be found. This is, in essence, a Branch and Bound[8] (BnB) algorithm [LD60].

We describe a basic algorithm for MAX-SAT, which is a DLL-style BnB algorithm. Most of the best performing exact algorithms for MAX-SAT are variants of that algorithm.

A BnB algorithm solving MAX-SAT explores the search tree induced by all the possible assignments in a depth-first manner. At each node, the algorithm compares the number of clauses unsatisfied by the best complete assignment found so far —called *Upper Bound* (UB)— with the *Lower Bound* (LB), the number of clauses unsatisfied by the current partial assignment plus an *underestimation* of the number of clauses that will become unsatisfied if we extend the current partial assignment to a complete assignment. Obviously, if $UB \leq LB$, a better assignment cannot be found from this point in the search (cf. Algorithm 2.5). In that case, the algorithm prunes the subtree below the current node and backtracks to a higher level in the search tree. If $UB > LB$, the current partial assignment is extended by instantiating one more variable p; which leads to create two branches from the current branch: the left branch amounts to solve the MAX-SAT instance that results to assign variable p to $false$, $\phi_{\neg p}$; and the right branch amounts to solve the MAX-SAT instance that results to assign variable p to $true$, ϕ_p. The solution to MAX-SAT is the value that the upper bound takes after exploring the entire search tree.

[7] More insights on MAX-SAT complexity can be found in [Kre88, AJ03].

[8] The term was created by Little et al. [LMSK63] for the Traveling Salesman problem. Soon after, a survey was published collecting all related methods solving combinatorial optimization problems [LW66].

In Algorithm 2.5, we use the following notation:

- `EmptyClauses`(ϕ) is a function that returns the number of empty clauses in ϕ.
- `LowerBound`(ϕ) is the sum of `EmptyClauses`(ϕ) plus an *underestimation*. The simplest lower bound incorporates no underestimation. Powerful lower bounds incorporate an underestimation easy to compute but able to estimate the number of unsatisfied clauses in an optimal solution of ϕ. In such a computation, the formula may be modified (as it actually happens in most of the lower bounds), but must be restored as soon as the computation is done in order to preserve the optimality of the solution.
- `InferenceRules`(ϕ) is a function that transforms a MAX-SAT instance into an equivalent MAX-SAT instance. The goal is to create an instance which is easier to solve. It is important to highlight that the modifications of the formula can be kept in the downward nodes of the search tree, in contrast with what happens in the lower bound computation.
- `UpperBound`(ϕ) is the number of unsatisfied clauses in the best solution found so far. The initial value is the number of clauses in the input MAX-SAT instance.
- `SelectVariable`(ϕ) is a function that heuristically selects an uninstantiated variable of ϕ.

Example 2.5 *Given the MAX-SAT instance*

$$(p_1 \vee p_2),\ \neg p_1,\ (p_1 \vee \neg p_2),\ (p_2 \vee p_3),\ (\neg p_2 \vee \neg p_3),\ p_2$$

we describe in Table 2.1 the application of algorithm MAX-SAT BnB: Every time a leaf in the search tree is reached, the upper bound is updated. In the current example, once a solution with 1 unsatisfied clause is found, most of the branches are pruned, the bound is reached. The search tree for this example is shown in Figure 2.2, where boxed figures represent partial solutions.

One of the most powerful techniques exploited in complete SAT algorithms is unit propagation. It is very useful for simplifying the SAT instance associated with each node of the search tree. As we show in the next example, unit propagation is not a sound inference rule in MAX-SAT.

Example 2.6 *Suppose we have the following CNF formula:*

$$\neg p_1,\ (p_1 \vee p_2),\ (p_1 \vee p_3),\ (\neg p_2 \vee \neg p_3),\ (p_1 \vee p_4),\ (p_1 \vee \neg p_4)$$

Applying unit propagation with unit clause $\neg p_1$ will bring a solution with 2 unsatisfied clauses, while the optimal solution has 1 unsatisfied clause (with the assignment $p_1 = true, p_2 = false, p_3 = false$).

Literal	Level	CNF formula	UB	LB
	Start	$p_1 \vee p_2, \neg p_1, p_1 \vee \neg p_2, p_2 \vee p_3, \neg p_2 \vee \neg p_3, p_2$	6	0
p_1	1	$\square, p_2 \vee p_3, \neg p_2 \vee \neg p_3, p_2$	6	1
p_2	2	$\square, \neg p_3$	6	1
p_3	3	$\square, \square$	6	2
	Leaf	UB updated to 2		
$\neg p_3$	3	$\square$	2	1
	Leaf	UB updated to 1		
$\neg p_2$	2	$\square, p_3, \square$	1	2
	Bound			
$\neg p_1$	1	$p_2, \neg p_2, p_2 \vee p_3, \neg p_2 \vee \neg p_3, p_2$	1	0
p_2	2	$\square, \neg p_3$	1	1
	Bound			
$\neg p_2$	2	$\square, p_3, \square$	1	2
	Bound			

Table 2.1: Execution track of a BnB for Example 2.5.

2.3.2 Local search and approximation algorithms for MAX-SAT

Local search algorithms for MAX-SAT use an objective function which is defined as the number of clauses that are satisfied under the given truth assignment. The general idea for solving MAX-SAT by local search is to perform a random walk in the search space which is biased in such a way that the number of satisfied clauses is maximized. In SAT, every time a satisfying assignment is found, the local search algorithm stops searching. Oppositely, a MAX-SAT local search algorithm never reaches such a situation, since there is no way to prove that a solution is optimal, unless traversing the whole search space.

The first known attempt to solve MAX-SAT by a local search algorithm is the Steepest Ascent-Mildest Descent approach due to Hansen and Jaumard [HJ90], that uses an underlying local search engine similar to GSAT incorporating tabu search and focusing only on MAX-SAT. Theoretical investigation on tabu search and MAX-SAT is described in [MG04, MG05].

Every local search algorithm for SAT can be used as a local search algorithm for MAX-SAT, since both try to minimize the number of unsatisfied clauses. There is an extended report on state-of-the-art local search algorithms for MAX-SAT in [SHR01]. Besides the solvers commented in such a report, it is worth to cite solvers that incorporate the last advances in local search for MAX-SAT: Telelis and Stamatopoulos [TS02], and Zang et al. [ZRL03] used an heuristic based on finding a pseudo-backbone; Smyth et al. [SHS03] used tabu search; Lardeux et al. [FL05] introduced three-valued variables.

Recent approximation algorithms used to compute upper bounds and lower

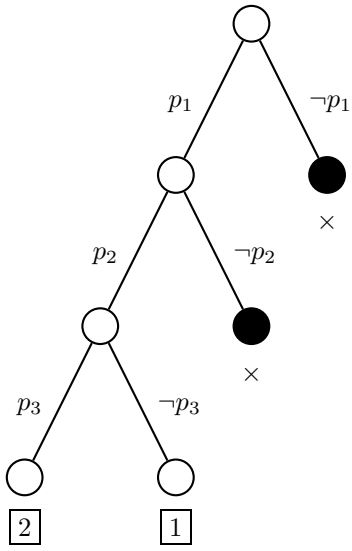


Figure 2.2: Search tree for MAX-SAT BnB applied to Example 2.5.

bounds for MAX-SAT are: van Maaren and van Norden [vMvN05] relaxed MAX-SAT to a sum of squares, and solved it using semidefinite programming; and Gomes et al. [GvHL06] relaxed MAX-SAT directly to semidefinite programming and solved it.

2.3.3 Overview of BnB algorithms for MAX-SAT

MAX-SAT has received the interest of a considerable number of researchers that have implemented many solvers with original techniques. To improve the performance of BnB MAX-SAT solvers, the design of algorithms has mainly focused on finding lower bounds that efficiently forecast the largest number of unsatisfied clauses, and inference rules that make the formula easier to solve.

We briefly present the lower bounds and the inference rules developed before and during the current research work. Our contributions on these topics are deeply analyzed in Chapter 3 and Chapter 4. Then, we describe the most representative variable selection heuristics. We conclude the section enumerating some extensions of MAX-SAT and weighted MAX-SAT.

Lower bounds

The first implemented exact algorithm solving MAX-SAT was due to Wallace and Freuder [WF96]. They applied their knowledge of Constraint Program-

ming[9] to MAX-SAT solving, and implemented a simple lower bound based on inconsistency counts:

$$\texttt{LB}_{IC}(\phi) = \texttt{EmptyClauses}(\phi) + \sum_{p \text{ occurs in } \phi} \min(\text{ic}(p), \text{ic}(\neg p)),$$

where ϕ is the CNF formula associated with the current partial assignment, and $\text{ic}(\ell)$ —inconsistency count of literal ℓ— is the number of clauses that would become unsatisfied if ℓ is satisfied; in other words, $\text{ic}(\ell)$ coincides with the number of unit clauses of ϕ that contain $\bar{\ell}$.

Shen and Zhang [ZSM03a, SZ04] defined a lower bound for MAX-2-SAT, called LB4, which detects disjoint inconsistent subformulas in MAX-2-SAT instances via linear unit resolution. This lower bound was implemented in a decision procedure with a static ordering of the variables.

Soon after, we developed a solver from scratch, Lazy [AMP04a, AMP04b, AMP05], within an original lower bound called Star. It detects disjoint inconsistent subformulas of the form: $l_1, l_2, \ldots, l_k, \bar{l}_1 \vee \bar{l}_2 \cdots \vee \bar{l}_k$. A description of Lazy is given in Chapter 5.

Xing and Zhang [XZ05] created a solver, MaxSolver, which uses an integer programming approach. A CNF formula is transformed into an integer programming formulation in order to efficiently compute a lower bound.

Then, we performed a powerful lower bound, UP [LMP05, LMP06], that detects disjoint inconsistent subformulas applying unit propagation. Such a lower bound is described in detail in Chapter 3.

Recently, Gomes et al. [GvHL06] have defined a lower bound based on semidefinite programming. They considered both the previous formulation of Xing and Zhang and the semidefinite relaxation for MAX-2-SAT proposed by Goemans and Williamson [GW95].

Inference rules

In Wallace and Freuder's algorithm [WF96], a similar strategy to that of forward checking [McG79] is applied, except that variables are fixed rather than reducing domains.The algorithm is as follows:

forall *variable* p *occurs in* ϕ **do**
  if $\mathit{EmptyClauses}(\phi) + \min(\text{ic}(p), \text{ic}(\neg p)) + |\text{ic}(p) - \text{ic}(\neg p)| \geq \mathit{UpperBound}(\phi)$ **then**
    if $\text{ic}(p) > \text{ic}(\neg p)$ **then** $\phi_{\neg p}$ **else** ϕ_p

When the first if-condition holds, the UC rule is applied to ϕ over the literal that brings the minimum number of inconsistencies.

The second implemented algorithm, and first available source code, was due to Borchers and Furman [BF99]. Although they implemented no underestimation, they incorporated two important techniques into MAX-SAT solving:

[9] Recently, it has been published an ACM survey [BHZ06] comparing constraint programming with SAT.

(i) the initial upper bound is computed with a local search algorithm, which allows them to solve MAX-SAT instances up to seven times faster than without that preprocessing; and (ii) when the difference between the lower bound and the upper bound is 1 ($UB - LB = 1$), unit propagation can be safely applied (in fact, when such a condition holds a SAT solver can be used to solve it). These two techniques were extended to weighted MAX-SAT.

Another important work, with a novel approach in MAX-SAT solving, is a set of articles due to Alber et al. [AGN01], Bansal and Raman [BR99] and Niedermeier et al. [NR00], which defined a set of MAX-SAT inference rules, created an algorithm and demonstrated its complexity.[10] In that algorithm, there is no underestimation. Gramm [GN00] implemented a MAX-2-SAT solver that incorporates the following rules: pure literal rule, Complementary Unit Clause (CUC) rule, restricted resolution rule[11], Almost Common Clauses (ACC) rule and three occurrences rule. This set of rules is very useful because either they fix the value of variables or reduce the arity of clauses, that makes a MAX-SAT instance to be easier to be solved. Most of such rules have been implemented in other solvers (e.g., Lazy, toolbar [LH05a], MaxSolver [XZ05] and MaxSatz [LMP05]).

Later, Shen and Zhang implemented a MAX-SAT solver [SZ05] with a static ordering of variables, which uses the implication graph and its strong connected components (SCC) in order to simplify a MAX-2-SAT instance, implementing two simplification techniques:

- If a SCC does not contain any conflicting literal, delete the clauses in the SCC from the original MAX-2-SAT formula.

- If there are more than one SCC, divide the original MAX-2-SAT formula according to the SCCs and run the MAX-2-SAT algorithm against each component separately.

Yet they found not useful those techniques because those situations are difficult to be found in a MAX-SAT instance. They also implemented ACC and restricted resolution rule as a preprocessing.

With a previous experience on solving Max-CSP [LMS99, LM02] and weighted CSP, de Givry et al. [dGLMS03] started to solve MAX-SAT by reducing it to weighted CSP and using the weighted CSP solver toolbar [LMS99]. Then, they developed a DLL-like BnB solver, inside the toolbar framework, equipped with efficient MAX-SAT inference rules [LH05a, HL06b].

Xing and Zhang [XZ05] created a solver, MaxSolver, using the inference rules pure literal, CUC and ACC, and defined the non-linear programming inference rule, which tries to fix variables. All these techniques were extended to weighted MAX-SAT.

[10] The most general time complexity is $\mathcal{O}(|\phi| \cdot 1.3803^{|\mathcal{C}|})$, where $|\phi|$ is the length of the CNF formula ϕ and $|\mathcal{C}|$ is the number of clauses in ϕ.

[11] Restricted resolution rule is the application of resolution in variables occurring exactly once positively and once negatively. Restricted resolution is called resolution in some papers that appeared before the definition of a sound and complete resolution rule for MAX-SAT.

Recently, there has been an effort to define a more general resolution rule for MAX-SAT. The first MAX-SAT resolution rule was defined in [LH05a], but the conclusions of the rules were not in clausal form. The definition of the conclusions in clausal form was done, independently, in [HL06b] and in [BLM06]. The completeness of the rule was proved in [BLM06]. From that completeness proof, it is easy to derive an exact variable elimination solver, which can be seen as an adaptation of DP [DP60] to MAX-SAT.

Variable selection heuristics

Most of the dynamic variable selection heuristics devoted to MAX-SAT are variants of the SAT heuristics MOMS and JW. We describe only the novel heuristics:

B+C Joy et al. [JMB97] designed a heuristic for a branch and cut algorithm[12] with a variable selection heuristic similar to JW:

$$J(\ell) = \sum_{\ell \in c} 1 - 2^{1-|c|}$$

This is the first heuristic devoted to MAX-SAT. In this heuristic, the larger a clause, the greater its influence, with the exception of unit clauses that have no influence.

AMP Alsinet et al. [AMP03a] proposed another modification of JW heuristic:

$$J(\ell) = \sum_{\ell \in c} \alpha(|c|)$$

where $\alpha(i)$ is the weight assigned to clauses of length i. Particularly, the weights given were $\alpha(1) = 1$, $\alpha(2) = 3$ and $\alpha(j) = 0.125$, if $j \geq 3$. The settings were determined experimentally.

MaxSolver Xing and Zhang [XZ05] implemented two variants of JW heuristic, for MAX-2-SAT:

$$J(\ell) = \sum_{\ell \in c} 5^{-|c|}$$

and for MAX-3-SAT:

$$J(\ell) = \sum_{\ell \in c} \beta(r)^{-|c|}$$

where β is a value that depends on the clause to variable ratio of the MAX-SAT instance. It varies from $\beta = 5$ to $\beta = 2$.

In general, we observe that the heuristics tend to reduce the influence of unit clauses, and increase the influence of binary clauses. All these heuristics were

[12] A branch and cut algorithm is similar to a branch and bound algorithm. Instead of applying inference rules, there is the addition of constraints or cuts. See [PR02] for an introductory survey.

extended to weighted MAX-SAT, multiplying each element in the addition by the weight of the clause. In the case of Jeroslow-Wang:

$$J(\ell) = \sum_{\ell \in c} w_i \cdot 2^{-|c|}$$

where w_i is the weight of clause i.

For static ordering variable selection heuristic, the solvers sort the variables by number of occurrences. We focus on the two available solvers that perform an additional procedure:

maxsat_LB4 Shen and Zhang [SZ05] used the created graph in order to improve their heuristic. A weight function is computed using the following procedure: At first each variable has a weight equal to 0. Then, they update the weight by finding the shortest path between every pair of $(l_1, \bar{l}_2)$ in a strongly connected component. If the path goes through node l_2 to l_3, they increase both the weights of l_2 and l_3 by 1. The experimental results in [SZ05] show that the ordering by the new weight function performs better than the occurrence ordering when the clause to variable ratio or the number of unsatisfied clauses is small.

Lazy Alsinet et al. [AMP04a] performed in Lazy a two-phase ordering variable heuristic: variables are ordered by number of occurrences and ties are broken depending on the variables that share the same clause. This heuristic will be described in detail in Chapter 5.

Extensions of MAX-SAT and weighted MAX-SAT

Finally, it is important to note that MAX-SAT and weighted MAX-SAT formalisms have been recently extended in several ways giving rise to new formalisms with more expressive power: multi-valued MAX-SAT [ADM+06, ABLM07], partial MAX-SAT [CIKM97, bM05, FM06, AM06b], quantified weighted MAX-SAT [Mal05], and Soft-SAT [AM06a]. There have been developed solvers implementing a branch and bound scheme and incorporating some of the MAX-SAT solving techniques for such formalisms.

2.3.4 Solvers submitted to the MAX-SAT Evaluation 2006

In the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-2006), following the steps of the SAT competition, the first MAX-SAT evaluation was hold.[13] A set of MAX-SAT and weighted MAX-SAT benchmarks and solvers were submitted and evaluated. The solvers submitted were:

- `ChaffBS` and `ChaffLS`, by Zhaoui Fu and Sharak Malik. It solves MAX-SAT. In order to translate a MAX-SAT instance into a MAX-SAT decision

[13] The results can be checked at `http://www.iiia.csic.es/maxsat06/` and in [ALMP08].

instance (a SAT instance), they append a distinct slack variable (or selector variable) to every MAX-SAT clause. A slack variable essentially means that the corresponding MAX-SAT clause can be left unsatisfied. Then, the problem can be: (i) asking if k clauses can be satisfied, or (ii) asking if *at least* k clauses can be satisfied (for further details refer to [FM06]).

- `MaxSatz`, by Li, Manyà and Planes. It solves MAX-SAT.
- `Lazy`, by Alsinet, Manyà and Planes. It solves MAX-SAT and weighted MAX-SAT.
- `SAT4jmaxsat`, by Le Berre. It solves MAX-SAT and weighted MAX-SAT. This solver uses a similar approach to the Chaff-based solvers, translating a MAX-SAT instance into a MAX-SAT decision instance and solving it with the SAT solver SAT4J. Such a solver is an implementation in Java of MiniSat [NE03].
- `Toolbar`, by de Givry, Heras, Larrosa and Shiex [HL06a]. It solves MAX-SAT and weighted MAX-SAT. It is actually a Weighted CSP solver, which offers the possibility of maintaining different forms of local consistency during search (e.g., full directional arc consistency).

2.4 Summary

This chapter gives an overview of techniques used in SAT and MAX-SAT solving. Firstly, some basic concepts commonly used in satisfiability solving have been introduced. Secondly, different techniques for SAT solving such as the DP algorithm and the DLL algorithm, recent efficient techniques in complete SAT solving, as well as representative local search algorithms are described. Thirdly, the branch and bound algorithm to solve MAX-SAT is also introduced, which is the basis for all discussions in the rest of the thesis, with special attention to the advances in lower bounds, inference rules and variables selection heuristics for MAX-SAT.

Chapter 3

Lower Bounds

A branch and bound algorithm solving MAX-SAT takes, at each node, the number of clauses unsatisfied by the best complete assignment found so far as an *upper bound* (UB), and the number of clauses unsatisfied by the current partial assignment plus an *underestimation* of the number of clauses that would become unsatisfied if we extend the current partial assignment to a complete assignment as a *lower bound* (LB). In this chapter, we focus on computing underestimations, although we call them lower bounds, making an abuse of language.

The quality of the lower bound has a great impact on the performance of branch and bound MAX-SAT solvers, because the better the lower bound, the more the search tree can be pruned. A good lower bound reaches early the upper bound in order to make the algorithm backtrack. The information gained in the lower bound computation cannot usually be kept throughout the search tree and needs to be recomputed at several nodes. Hence, the lower bound should be both as large and as cheap to compute as possible.

This chapter is structured as follows. In Section 3.1, we review some state-of-the-art lower bounds. In Section 3.2, we introduce the star rule, which is our first original lower bound. In Section 3.3, we define three original lower bounds that detect inconsistent disjoint subformulas by applying unit propagation. In Section 3.4, we improve the previous lower bound in two ways: by improving the manner the unit clauses are chosen in unit propagation; and by adding failed literal detection. We conclude the chapter in Section 3.5 reporting on the empirical evaluation.

3.1 Related work

In this section we review some of the most relevant lower bounds defined for MAX-SAT in the literature. We illustrate the behavior of the lower bounds using the following CNF formula:

$$\phi = \{\ \square,\quad \neg p_1, p_1,\quad \neg p_2, (p_2 \vee p_3), (p_2 \vee \neg p_3),\quad \neg p_4, \neg p_5, p_6, (p_4 \vee p_5 \vee \neg p_6)\ \} \tag{3.1}$$

Observe that the minimum number of unsatisfied clauses in ϕ is 4. For clarity, we placed a space between each disjoint unsatisfiable subformula.

No underestimation The simplest method for computing lower bounds consists of counting the number of clauses unsatisfied by the current partial assignment (empty clauses) without considering any underestimation.

Given the CNF formula ϕ in Equation 3.1, the lower bound computed by this method is 1.

Inconsistency Counts (IC) The following lower bound, based on inconsistency counts, was defined by Wallace and Freuder [WF96] :

$$\mathtt{LB}_{IC}(\phi) = \mathit{EmptyClauses}(\phi) + \sum_{p \text{ occurs in } \phi} \min(\mathrm{ic}(p), \mathrm{ic}(\neg p)),$$

where ϕ is the CNF formula associated with the current partial assignment, $\mathit{EmptyClauses}(\phi)$ is the number of empty clauses derived so far, and $\mathrm{ic}(\ell)$ —inconsistency count of literal ℓ— is the number of clauses that would become unsatisfied if ℓ is satisfied; in other words, $\mathrm{ic}(\ell)$ coincides with the number of unit clauses of ϕ that contain $\bar{\ell}$.

Given the CNF formula ϕ in Equation 3.1,

$$\mathtt{LowerBoundIC}(\phi) = 2.$$

Algorithm 3.1: `LowerBoundIC`(ϕ) : Computation of lower bound inconsistency count

```
Function LowerBoundIC(φ : CNF formula) : Natural
    Data: U_ℓ : set of unit clauses in φ containing ℓ
    underestimation ← EmptyClauses (φ)
    forall variable p ∈ Var(φ) do
        ▷ Computation of IC for unassigned variable p ◁
        while U_p ≠ ∅ ∧ U_¬p ≠ ∅ do
            U_p ← U_p \ (p)
            U_¬p ← U_¬p \ (¬p)
            underestimation ← underestimation + 1
    return underestimation
```

Time complexity : $\mathcal{O}(|\phi|)$

The application of the lower bound is shown in Algorithm 3.1. In the analysis of the algorithm, we assume that, for each literal there is a list of the binary clauses in which the literal occurs. For each variable, we have to count the number of unit clauses for the positive literal, and the number of unit clauses for the negative literal. Since the algorithm has to deal with all the literals in the worst case, its cost is in $\mathcal{O}(|\phi|)$.

LB4 Shen and Zhang [SZ04] defined a lower bound for MAX-2-SAT, called LB4: they detect disjoint inconsistent subformulas in MAX-2-SAT instances via linear unit resolution (cf. Algorithm 3.2). They assume a static ordering of the variables.

Given the CNF formula ϕ in Equation 3.1,

$$\texttt{LowerBoundLB4}(\phi) = 3.$$

Notice that the algorithm only considers unary and binary clauses.

Algorithm 3.2: `LowerBoundLB4`(ϕ) : Computation of lower bound LB4

Function `LowerBoundLB4`(ϕ *: CNF formula*) **:** ***Natural***
- **Data**: $\mathcal{U}_\ell$: set of unit clauses in ϕ containing ℓ
- underestimation $\leftarrow$ `EmptyClauses` (ϕ)
- **forall** *variable* $p \in Var(\phi)$ **do**
  - **forall** *literal* $l_1 \in \{p, \neg p\}$ **do**
    - underestimation $\leftarrow$ underestimation + $\min(ic(p), ic(\neg p))$
    - **forall** *binary clause* $(\bar{l}_1 \vee l_2) \in \phi$ **do**
      - ▷ Assuming that the variable in l_2 is assigned later than the variable in l_1 ◁
      - **if** $\mathcal{U}_{l_1} \neq \emptyset$ **then**
        - $\mathcal{U}_{l_1} \leftarrow \mathcal{U}_{l_1} \setminus (l_1)$
        - $\mathcal{U}_{l_2} \leftarrow \mathcal{U}_{l_2} \cup (l_2)$
- **return** underestimation

Time complexity : $\mathcal{O}(|\phi|)$

In the analysis of Algorithm 3.2, we assume that, for each literal, there is a counter of the number of unit clauses containing that literal, and a list of the clauses in which the literal occurs. The operations in each loop can be performed in constant time, and each loop is executed at most m times, where m is the number of clauses. So, the complexity of the three loops is in $\mathcal{O}(|\phi|)$.

Integer programming based lower bound Xing and Zhang [XZ05] use an integer programming approach to compute a fast lower bound. A CNF formula is transformed into an integer programming (IP) formulation, and rather than solving it with IP, it allows the variables to take a continuous value in the range $[0 - 1]$. This makes the computation faster. When the values of the variables obtained are close to 0 or 1, they are approximated to the closest value. But when the value is close to 1/2, it yields no information. In order to avoid this situation, they restrict the application of the computation to the partial formulas having unit clauses. Xing and Zhang observed that such a lower bound is not efficient when the problem is underconstrained.

Given the CNF formula ϕ in Equation 3.1,

$$\texttt{LowerBound}_{IP}(\phi) = 4.$$

Semi-Definite Programming Gomes et al. [GvHL06] have recently defined a lower bound based on semidefinite programming.[1] They considered both the previous formulation of Xing and Zhang and the semidefinite relaxation for MAX-2-SAT proposed by Goemans and Williamson [GW95]. In both cases, they apply the following scheme:

1. solve the relaxation
2. order the variables by their absolute value
3. fix the first n variables (this value is an algorithm parameter)
4. extend the partial assignment to a total assignment.

The lower bound defined by Gomes et al. experimentally outperforms the theoretical approximation of 0.940 [LLZ02] for random MAX-2-SAT.

3.2 Star rule

Our first original lower bound is the *star rule* [AMP04a], in which the underestimation of the lower bound is the number of disjoint inconsistent subformulas of the form

$$\{l_1, \ldots, l_k, \bar{l}_1 \vee \cdots \vee \bar{l}_k\}.$$

In contrast to previous lower bounds, clauses with size greater than 2 can be considered to compute underestimations.

When $k = 1$, the star rule becomes $\texttt{LB}_{IC}$; i.e., the star rule subsumes the lower bound based on inconsistency counts. When $k = 2$, the star rule detects inconsistencies that are also detected by LB4. Nevertheless, the star rule and LB4 cannot be compared because LB4 detects inconsistencies like $\{l_1, \neg l_1 \vee l_2, \neg l_2 \vee l_3, \neg l_3\}$ which are beyond the reach of the star rule, and the star rule detects inconsistencies like $\{l_1, l_2, l_3, \neg l_1 \vee \neg l_2 \vee \neg l_3\}$ which are beyond the reach of LB4.

In Algorithm 3.3, we show the algorithm as it was implemented in [AMP04a]: It only deals with $k = 2$ with a static variable ordering. In this solver, the algorithm can be efficiently implemented with cost $\mathcal{O}(|\phi|)$.

Given the CNF formula ϕ in Equation 3.1, Algorithm 3.3 computes $\texttt{StarRule}(\phi) = 2$. Although the general star rule computes $\texttt{StarRule}(\phi) = 3$.

[1] A survey of SDP-based approximation algorithms for the MAX-SAT problem is introduced in [Anj05].

Algorithm 3.3: `StarRule`(ϕ) : Computation of lower bound star rule

```
Function StarRule(φ : CNF formula) : Natural
    Data: U_ℓ(φ) : set of unit clauses in φ containing ℓ
    underestimation ← LowerBoundIC (φ)
    forall variable p_1 ∈ Var(φ) do
        forall literal l_1 ∈ {p_1, ¬p_1} do
            if U_{l_1}(φ) ≠ ∅ then
                forall variable p_2 ∈ Var(φ) further than p_1 do
                    forall literal l_2 ∈ {p_2, ¬p_2} do
                        if U_{l_2}(φ) ≠ ∅ ∧ (l̄_1 ∨ l̄_2) ∈ φ then
                            φ ← φ \ {l_1, l_2, (l̄_1 ∨ l̄_2)}
                            underestimation ← underestimation + 1
    return underestimation
```

Time complexity : $\mathcal{O}(|\phi|)$

3.3 Lower Bound UP

Unit propagation is one of the most powerful inference techniques in SAT solving. Even when it cannot be used like in SAT because its application can lead to non-optimal solutions, we realized that it can be used to compute an underestimations of the lower bound in branch and bound MAX-SAT solvers.

Lower bound UP works as follows: Once a contradiction in a CNF formula ϕ is detected via unit propagation, UP identifies a subset of clauses ϕ' involved in that unit propagation from which we can derive a unit refutation.[2] Then, $\phi := \phi - \phi'$, and the underestimation is increased by one. This process is repeated until no more contradictions can be derived by unit propagation. Observe that ϕ must contain unit clauses for applying unit propagation.

Example 3.1 *Let ϕ be the following CNF formula:*

$$p_1, p_2, p_3, p_4, (\neg p_1 \vee \neg p_2 \vee \neg p_3), (\neg p_1 \vee p_6), \neg p_4, p_5, (\neg p_5 \vee \neg p_2), (\neg p_5 \vee p_2).$$

Using lower bound UP, we are able to establish that the number of unsatisfied clauses in ϕ is at least 3. The steps performed are the following ones:

1. *$\phi' = \{p_1, p_2, p_3, \neg p_1 \vee \neg p_2 \vee \neg p_3\}$, $\phi = \{\neg p_1 \vee p_6, p_4, \neg p_4, p_5, \neg p_5 \vee \neg p_2, \neg p_5 \vee p_2\}$, and the underestimation is 1. Observe that $\neg p_1 \vee p_6$ is involved in the*

[2] *Unit resolution* states that from ℓ and $\neg\ell \vee D$, where ℓ is a literal and D is a disjunction of literals, we can derive resolvent D.

A *derivation* (or *proof*) of a clause c from a clause set ϕ is a sequence $c_1, c_2, \ldots, c_m$ of clauses such that c_m is the clause c, and for every $i = 1, \ldots, m$, c_i is either a clause in ϕ or a resolvent of two clauses c_a, c_b with $a, b < i$.

A *refutation* is a derivation having as conclusion the empty clause. A *unit refutation* is a refutation in which all resolvents are derived with unit resolution.

unit propagation in which we have detected a contradiction, but it is not included in ϕ' because it is not needed to derive a unit refutation.

2. *$\phi' = \{p_4, \neg p_4\}$, $\phi = \{\neg p_1 \vee p_6, p_5, \neg p_5 \vee \neg p_2, \neg p_5 \vee p_2\}$, and the underestimation is 2.*

3. *$\phi' = \{p_5, \neg p_5 \vee \neg p_2, \neg p_5 \vee p_2\}$, $\phi = \{\neg p_1 \vee p_6\}$, and the underestimation is 3.*

Lower bound UP captures lower bound IC: if we apply UP to the CNF formula from Example 3.1, it is captured in step 2, when we derive the empty clause from p_4 and $\neg p_4$. Observe that while UP returns 3, lower bound IC returns 1.

Lower bound UP also captures the star rule. In the CNF formula from Example 3.1, the star rule is captured in step 1, when we derive the empty clause from $p_1, p_2, p_3, (\neg p_1 \vee \neg p_2 \vee \neg p_3)$, and in step 2, when we derive the empty clause from p_4 and $\neg p_4$. While UP returns 3, the star rule returns 2.

Moreover, UP captures inconsistencies which are beyond the reach of lower bounds like IC and the star rule. For example, when UP derives a refutation from $p_5, (\neg p_5 \vee \neg p_2), (\neg p_5 \vee p_2)$ in the CNF formula from Example 3.1.

3.3.1 Understanding the lower bound through the implication graph

Given a CNF formula with many clauses and conflicts, it becomes a difficult task to see which clauses will be chosen by lower bound UP. For the sake of better understanding lower bound UP, and how it creates a refutation, we introduce the concept of *implication graph*.

Following the definition in [BKS04], an implication graph G at a given stage of DLL is a directed acyclic graph. It is constructed as follows:

1. Create a node for each unit clause, labeled with its literal. These will be the indegree zero root nodes of G.

2. While there exists a clause $c = (l_1 \vee \cdots \vee l_k \vee l_{k+1})$ such that $\neg l_1, \ldots, \neg l_k$ label nodes in G,

   (a) Add a node labeled l_{k+1} if not already present in G.

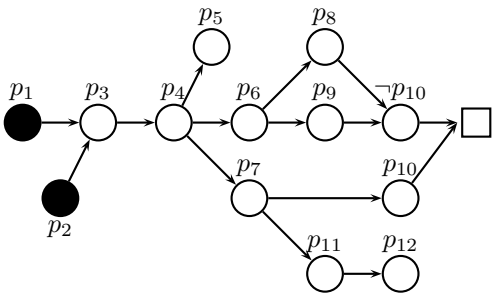
   (b) Add edges (l_i, l_{k+1}), $1 \leq i \leq k$, if not already present.

3. Add to G a special node $\square$. For any variable p which occurs both positively and negatively in G, add directed edges from p and $\neg p$ to $\square$.

Example 3.2 *Let ϕ be the following CNF formula:*

$$(\neg p_8 \vee \neg p_9 \vee \neg p_{10}), (\neg p_1 \vee \neg p_2 \vee p_3), (\neg p_6 \vee p_9), (\neg p_6 \vee p_8), (\neg p_7 \vee p_{10}),$$
$$(\neg p_4 \vee p_7), (\neg p_4 \vee p_6), (\neg p_4 \vee p_5), (\neg p_7 \vee p_{11}), (\neg p_{11} \vee p_{12}), (\neg p_3 \vee p_4),$$
$$(\mathbf{p_1}), (\mathbf{p_2})$$

Following the steps for the creation of its implication graph, we obtain the graph below. In the following, each node is labeled with the literal associated with its unit clause, a square means an empty clause, and black nodes mean starting unit clauses.



In order to derive a unit refutation, there is no need to select all the clauses involved in an implication graph. Only the nodes having a path to the conflict are needed. We introduce here the concept of *conflict graph* [BKS04]. A conflict graph H is any subgraph of an implication graph with the following properties:

1. H contains $\square$ and exactly one conflict variable.

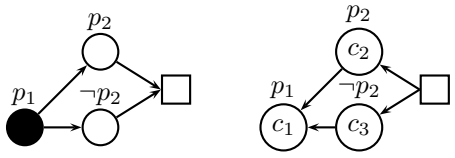
2. All nodes in H have a path to $\square$.

3. Every node ℓ in H other than $\square$ either corresponds to a starting unit clause or has precisely the nodes $\neg l_1, \neg l_2, \ldots, \neg l_k$ as predecessors where $(l_1 \vee l_2 \vee \cdots \vee l_k \vee \ell)$ is a known clause.

While an implication graph may or may not contain conflicts, a conflict graph always contains exactly one. The set of nodes of a conflict graph represent an inconsistent formula.

In the creation of graph G, we are interested in obtaining a conflict graph. Every time a new node is created, `UnitPropagation` (cf. Algorithm 3.5) saves the path backwards to the preceding unit clauses, as is shown in Example 3.3. The implementation of such a path is called an Implemented Graph.

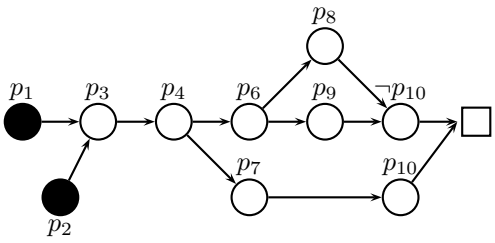
Example 3.3 *Given the formula ϕ $c_1 : p_1, c_2 : (\neg p_1 \vee p_2), c_3 : (\neg p_1 \vee \neg p_2)$, its representations is shown below: (a) as an implication graph and (b) as an implemented graph.*



(a) *(b)*

Once the implication graph is constructed, it is easy to identify all the nodes from which there exists a path to the contradiction. Starting from the empty clause, the set of nodes in the path backwards to the root nodes have to be collected. This makes the nodes that are not in the refutation to be never reached.

Example 3.4 *Taking the implication graph in Example 3.2, the refutation is formed by all the nodes excluding the nodes having no path to the contradiction, as shown below:*



In this case, clauses $(\neg p_7 \vee p_{11})$, $(\neg p_{11} \vee p_{12})$ *and* $(\neg p_4 \vee p_5)$ *have been removed.*

3.3.2 Implementing the lower bound UP

The lower bound computation in detail is as follows: If there exists any unit clause, apply unit propagation until a contradiction is found. In such a case, remove the set of inconsistent clauses and start the process again. This computation is shown in Algorithm 3.4, where `UnitPropagation` returns the set of clauses deriving an empty clause if any; otherwise, the function returns the empty set.

It is known that the standard unit propagation has a linear time complexity in the length of the formula, $\mathcal{O}(|\phi|)$ [GEI91, Fre95]. The loop iterates k steps (it detects k inconsistent subformulas) plus an additional step if there is any unit clause left. As a whole, the lower bound computation has a time complexity in $\mathcal{O}(k \cdot |\phi|)$. In the worse case, every unit clause may derive an empty clause. So, the time complexity is also $\mathcal{O}(|\mathcal{U}| \cdot |\phi|)$, where $|\mathcal{U}|$ is the number of unit clauses in ϕ.

As commented above, `UnitPropagation`(ϕ) (cf. Algorithm 3.5) implements the application of unit propagation. This function returns the set of clauses deriving an empty clause if any; otherwise, the function returns $\emptyset$. Inside the algorithm, `GetDerivation` collects the sequence of clauses deriving the empty clause; i.e., the nodes in the conflict graph. Regarding the analysis of Algorithm 3.5, `UnitPropagation` has linear time complexity in the length of the formula $\mathcal{O}(|\phi|)$, in the worst case. Therefore, adding Line UNION to `UnitPropagation` has no influence on the complexity, because it can be efficiently implemented as an assignment with constant cost. Thus, the loop has time complexity $\mathcal{O}(|\phi|)$. The recursive function `GetDerivation` has time cost $\mathcal{O}(\min(m, n))$, because the

Algorithm 3.4: `LowerBoundUP`(ϕ) : Computation of lower bound UP

```
Function LowerBoundUP(φ : CNF formula) : Natural
    Data: φ : set of clauses in the derivation of a conflict, or the empty
          set if no conflict is found
    underestimation ← EmptyClauses(φ)
    finished ← false
    while ∃ unit clause in φ ∧ ¬ finished do
        φ ← UnitPropagation(φ)
        φ ← φ \ φ
        ▷ Remove the clauses in the refutation ◁
        if φ ≠ ∅ then
            underestimation ← underestimation + 1
        else
            finished ← true
    return underestimation
```

Time complexity : $\mathcal{O}(k \cdot |\phi|)$

implication graph has two implicit constraints: a maximum of n nodes, there cannot be more nodes than variables in the CNF formula; and a maximum of m nodes, there cannot be more nodes than clauses in the CNF formula. Therefore, the overall complexity of the algorithm is $\mathcal{O}(|\phi|)$, observing that $|\phi|$ is a generous upper bound for `GetDerivation`.

3.4 UP improved: Choosing the best unit clause

We propose two new lower bounds, called UP* and UP^S, which improve UP by using a different heuristic for propagating unit clauses in unit propagation. Then, we propose three new lower bounds, called UP$_{FL}$, UP$^*_{FL}$ and UP$^S_{FL}$, which are, respectively, extensions of UP, UP* and UP^S incorporating the detection of failed literals.

3.4.1 Lower bounds improving UP

UP gives an underestimation of the number of disjoint inconsistent subformulas in a CNF formula ϕ using unit propagation, which means that (i) each inconsistent subformula contains at least one unit clause and, therefore, the number of detected inconsistencies is bounded by the number of unit clauses in ϕ; and (ii) clauses in an inconsistent subformula cannot be used to derive other inconsistent subformulas.

In order to define better orderings than the one implemented in lower bound UP for propagating unit clauses in `UnitPropagation`, the goal is twofold: (i) we need to find disjoint inconsistent subformulas containing as few unit clauses as possible, leaving more unit clauses in the remaining formula to derive further

Algorithm 3.5: `UnitPropagation`(ϕ) : Application of unit propagation for lower bound UP

```
Function UnitPropagation(φ : CNF formula) : Set of clauses
    Data: G : Implemented graph initialized to empty
    φ' ← φ
    while ∃ unit clause in φ' ∧ □ ∉ φ do
NextUC      c ← NextUnitClause(φ')
            ▷ Assuming c = {ℓ}                                   ◁
Union       G ← G ∪ (c, ℓ)
            φ' ← UnitClauseRule(φ', ℓ)
    if □ ∈ φ' then return GetDerivation(□, φ, G)
    else return ∅

Time complexity : O(|φ|)
```

inconsistent subformulas; and (ii) each inconsistent subformula should also contain as few non-unit clauses as possible. As a result, we provide two new lower bounds: UP* and UP^S. We will investigate the differences between the three different heuristics for choosing unit clauses.

Lower bound UP stores unit clauses in a queue, so that older unit clauses are preferred to more recent unit clauses when using them. We define a new lower bound, UP^S, that stores all unit clauses in a stack S, so that the last inserted unit clause is the first used. The last lower bound, UP*, maintains two queues: Q_1 and Q_2. When UP* starts to search for an inconsistent subformula, Q_1 contains all the unit clauses of the CNF formula under consideration (more recently derived unit clauses are at the end of Q_1), and Q_2 is empty. The unit clauses derived during the application of unit propagation are stored in Q_2, and unit propagation does not use any unit clause from Q_1 unless Q_2 is empty. In other words, UP creates the implication graph in a breadth-first manner, UP^S in a depth-first manner, and UP* in a kind of locality and breadth-first manner.

The three different lower bounds are implemented as a different heuristic for the function `NextUnitClause`, in line NextUC of Algorithm 3.5.

Examples with lower bounds UP, UP^S and UP*

For the sake of better understanding the three lower bounds, we provide three examples, to illustrate the different behaviour of each lower bound. In the examples provided, we observe that: UP consumes more unit clauses than UP^S and UP^*; UP^S consumes more clauses than UP^*; and UP^* consumes fewer unit clauses than UP^S.

Example 3.5 *Let ϕ_1 be the MAX-SAT instance $\{p_1, p_2, p_3, \neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2\}$. We show that UP detects exactly one inconsistent subformula while UP* and UP^S are able to detect two inconsistent subformulas.*

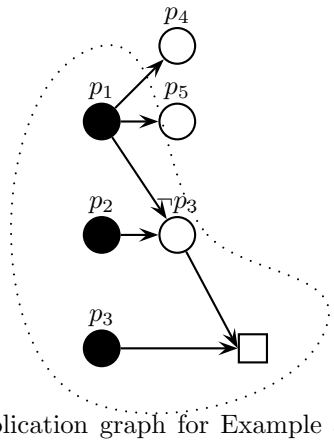
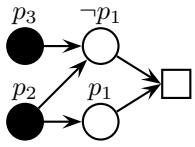


Figure 3.1: Created implication graph for Example 3.5 applying lower bound UP. The dotted area contains the conflict graph.

UP: Initially, $Q = [p_1, p_2, p_3]$. *When* p_1 *is propagated, unit clauses* p_4 *and* p_5 *are added to* Q *(*$Q = [p_2, p_3, p_4, p_5]$*), clause* $p_1 \vee \neg p_2$ *is removed, and clause* $\neg p_1 \vee \neg p_2 \vee \neg p_3$ *becomes* $\neg p_2 \vee \neg p_3$. *When* p_2 *is propagated,* $\neg p_2 \vee \neg p_3$ *becomes* $\neg p_3$, *which is added to* Q *(*$Q = [p_3, p_4, p_5, \neg p_3]$*). When* p_3 *is propagated, the empty clause is derived. The inconsistent subformula detected by UP is* $\{p_1, p_2, p_3, \neg p_1 \vee \neg p_2 \vee \neg p_3\}$. *The remaining clauses* $\{\neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5, p_1 \vee \neg p_2\}$ *do not contain any unit clause and, therefore, UP stops.*

Queue	
$\mathbf{p_1}, p_2, p_3$	$\neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2$
$\mathbf{p_2}, p_3, p_4, p_5$	$\neg p_4 \vee \neg p_5, \neg p_2 \vee \neg p_3$
$\mathbf{p_3}, p_4, p_5, \neg p_3$	$\neg p_4 \vee \neg p_5$
p_4, p_5	$\{p_1, p_2, p_3, \neg p_1 \vee \neg p_2 \vee \neg p_3\} \vdash \square$
$\varnothing$	$\neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5, p_1 \vee \neg p_2$

UP^S: Initially, $S = [p_3, p_2, p_1]$ *(we assume* p_1 *is at the bottom of the stack). When* p_3 *is propagated, clause* $\neg p_1 \vee \neg p_2 \vee \neg p_3$ *becomes* $\neg p_1 \vee \neg p_2$, *and* $S = [p_2, p_1]$. *When* p_2 *is propagated, unit clauses* $\neg p_1$ *and* p_1 *are added to* S *(*$S = [p_1, \neg p_1, p_1]$*). When* p_1 *is propagated, the empty clause is derived. The first inconsistent subformula detected is* $\{p_2, p_3, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2\}$. *Next,* UP^S *derives another contradiction from the remaining clauses:* $\{p_1, \neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5\}$. *Now,* $S = [p_1]$. *When* p_1 *is propagated, unit clauses* p_4 *and* p_5 *are added to* S *(*$S = [p_5, p_4]$*). When* p_4 *is propagated, unit clause* $\neg p_5$ *is added to* S *(*$S = [\neg p_5, p_5]$*). When* $\neg p_5$ *is propagated, the empty clause is derived. The second inconsistent subformula is* $\{p_1, \neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5\}$.



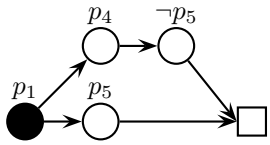
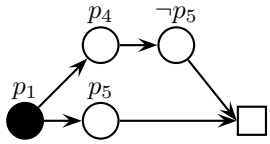


Figure 3.2: Created implication graphs for Example 3.5 applying lower bound UP^S. Both graphs correspond to the conflict graphs.



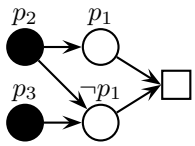


Figure 3.3: Created implication graphs for Example 3.5 applying lower bound UP^*. Both graphs correspond to the conflic graphs.

Stack	
$\mathbf{p_3}, p_2, p_1$	$\neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2$
$\mathbf{p_2}, p_1$	$\neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5, \neg p_1 \vee \neg p_2, p_1 \vee \neg p_2$
$\mathbf{p_1}, \neg p_1, p_1$	$\neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5$
p_4, p_5	$\{p_2, p_3, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2\} \vdash \square$
$\mathbf{p_1}$	$\neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5$
$\mathbf{p_4}, p_5$	$\neg p_4 \vee \neg p_5$
$\mathbf{p_5}, \neg p_5$	$\emptyset$
$\varnothing$	$\{p_1, \neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5\} \vdash \square$

UP^*: Initially, $Q_1 = [p_1, p_2, p_3]$. When p_1 is propagated, unit clauses p_4 and p_5 are added to Q_2 ($Q_2 = [p_4, p_5]$), clause $p_1 \vee \neg p_2$ is removed, and clause

$\neg p_1 \vee \neg p_2 \vee \neg p_3$ becomes $\neg p_2 \vee \neg p_3$. We then propagate p_4 and derive $\neg p_5$, which is added to Q_2 ($Q_2 = [p_5, \neg p_5]$). When p_5 is propagated, the empty clause is derived. The first inconsistent subformula detected is $\{p_1, \neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5\}$. Observe that UP consumed exactly one unit clause from the input formula. Next, UP* detects another contradiction in the remaining clauses: $\{p_2, p_3, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2\}$. Now, $Q_1 = [p_2, p_3]$. When p_2 is propagated, unit clauses p_1 is added to Q_2 ($Q_2 = [p_1]$) and clause $\neg p_1 \vee \neg p_2 \vee \neg p_3$ becomes $\neg p_1 \vee \neg p_3$. When p_1 is propagated, unit clauses $\neg p_3$ is added to Q_2 ($Q_2 = [\neg p_3]$). When $\neg p_3$ is propagated, the empty clause is derived. The second inconsistent subformula is $\{p_2, p_3, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2\}$.*

Q_1	Q_2	
$\mathbf{p_1}, p_2, p_3$	$\varnothing$	$\neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2$
p_2, p_3	$\mathbf{p_4}, p_5$	$\neg p_4 \vee \neg p_5, \neg p_2 \vee \neg p_3$
p_2, p_3	$\mathbf{p_5}, \neg p_5$	$\neg p_2 \vee \neg p_3$
p_2, p_3	$\varnothing$	$\{p_1, \neg p_1 \vee p_4, \neg p_1 \vee p_5, \neg p_4 \vee \neg p_5\} \vdash \square$
$\mathbf{p_2}, p_3$	$\varnothing$	$\neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2$
p_3	$\mathbf{p_1}$	$\neg p_1 \vee \neg p_3$
p_3	$\mathbf{\neg p_3}$	$\emptyset$
$\varnothing$	$\varnothing$	$\{p_2, p_3, \neg p_1 \vee \neg p_2 \vee \neg p_3, p_1 \vee \neg p_2\} \vdash \square$

Example 3.5 suggests that one of the drawbacks of UP is that it consumes unit clauses from the input formula that could be avoided, which is a direct consequence of the ordering in which unit clauses are propagated.

Example 3.6 *Let ϕ_2 be the MAX-SAT instance $\{p_1, \neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3, \neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9\}$. We show that, in this case, UP^S consumes more clauses (not necessarily unit clauses) than UP* when detecting inconsistent subformulas.*

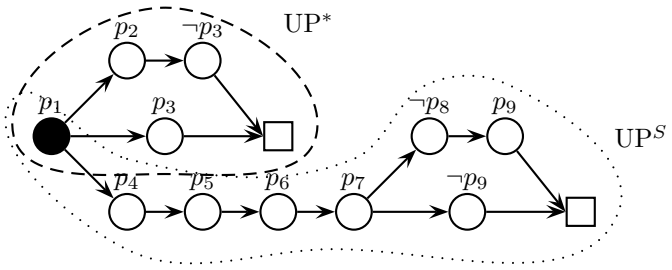


Figure 3.4: Implication graph for Example 3.6. The dotted area contains the conflict graph nodes detected by UP^S; and the dashed area contains the conflict graph nodes detected by UP*.

UP^S: Initially, $S = [p_1]$. When p_1 is propagated, unit clauses p_2, p_3, and p_4 are added to S ($S = [p_4, p_3, p_2]$). When p_4 is propagated, unit clause p_5 is added to S ($S = [p_5, p_3, p_2]$). When p_5 is propagated, unit clause p_6 is added to S ($S = [p_6, p_3, p_2]$). When p_6 is propagated, unit clause p_7 is added to S ($S = [p_7, p_3, p_2]$). When p_7 is propagated, unit clauses $\neg p_8$ and $\neg p_9$ are added to S ($S = [\neg p_9, \neg p_8, p_3, p_2]$). When $\neg p_9$ is propagated, unit clause p_8 is added to S. ($S = [p_8, \neg p_8, p_3, p_2]$). When p_8 is propagated, the empty clause is derived. The inconsistent subformula detected by UP^S is $\{p_1, \neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9\}$, which contains 8 clauses.

Stack	
$\mathbf{p_1}$	$\neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3, \neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6,$ $\neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\mathbf{p_4}, p_3, p_2$	$\neg p_2 \vee \neg p_3, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8,$ $\neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\mathbf{p_5}, p_3, p_2$	$\neg p_2 \vee \neg p_3, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\mathbf{p_6}, p_3, p_2$	$\neg p_2 \vee \neg p_3, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\mathbf{p_7}, p_3, p_2$	$\neg p_2 \vee \neg p_3, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\neg\mathbf{p_8}, \neg p_9, p_3, p_2$	$\neg p_2 \vee \neg p_3, p_8 \vee p_9$
$\mathbf{p_9}, \neg p_9, p_3, p_2$	$\neg p_2 \vee \neg p_3$
$\neg p_9, p_3, p_2$	$\{p_1, \neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8,$ $\neg p_7 \vee \neg p_9, p_8 \vee p_9\} \vdash \square$
	$\neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3$

UP^*: Initially, $Q_1 = [p_1]$ and Q_2 are empty. When p_1 is propagated, unit clauses p_2, p_3, and p_4 are added to Q_2 ($Q_2 = [p_2, p_3, p_4]$). When p_2 is propagated, unit clause $\neg p_3$ is added to Q_2 ($Q_2 = [p_3, p_4, \neg p_3]$). When p_3 is propagated, the empty clause is derived. The inconsistent subformula detected by UP^* is $\{p_1, \neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3\}$, which contains 4 clauses.

Q_1	Q_2	
$\mathbf{p_1}$	$\varnothing$	$\neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3, \neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6,$ $\neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\varnothing$	$\mathbf{p_2}, p_3, p_4$	$\neg p_2 \vee \neg p_3, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8,$ $\neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\varnothing$	$\mathbf{p_3}, p_4, \neg p_3$	$\neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\varnothing$	p_4	$\{p_1, \neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3\} \vdash \square$
$\varnothing$	$\varnothing$	$\neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8,$ $\neg p_7 \vee \neg p_9, p_8 \vee p_9$

Example 3.6 suggests that UP^S tends to find larger inconsistent subformulas than UP^*; i.e., UP^S can consume more clauses than UP^* to derive an empty

clause. This is so because UP^S, when there are several possibilities of deriving an empty clause from a unit clause, just finds the first derivation, while it can be shown that UP^* tends to find shorter derivations. UP^* makes one step in each possible derivation in parallel, stopping all derivations when the first empty clause is found. In other words, UP^S performs a depth-first search while UP^* performs a breadth-first search.

Example 3.7 *Let clauses in ϕ_2 be ordered as follows: $\{p_1, \neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9, \neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3\}$. We show that UP^* finds a derivation as short as UP^S.*

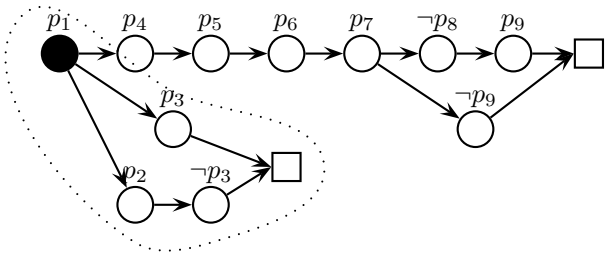


Figure 3.5: Implication graph for Example 3.7. The dotted area contains the conflict graph nodes detected by UP^S and UP^*.

In this case, UP^S finds the shortest derivation of an empty clause, because the shortest derivation happens to be the first one. However, UP^* always finds this derivation in the following way: initially, $Q_1 = [p_1]$ and Q_2 is empty. When p_1 is propagated, unit clauses p_4, p_2, and p_3 are added to Q_2 ($Q_2 = [p_4, p_2, p_3]$). When p_4 is propagated, unit clause p_5 is added to Q_2 ($Q_2 = [p_2, p_3, p_5]$). When p_2 is propagated, unit clause $\neg p_3$ is added to Q_2 ($Q_2 = [p_3, p_5, \neg p_3]$). When p_3 is propagated, the empty clause is derived.

Q_1	Q_2	
$\mathbf{p_1}$	$\varnothing$	$\neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9, \neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3$
$\varnothing$	$\mathbf{p_4}, p_2, p_3$	$\neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9, \neg p_2 \vee \neg p_3$
$\varnothing$	$\mathbf{p_2}, p_3, p_5$	$\neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9, \neg p_2 \vee \neg p_3$
$\varnothing$	$\mathbf{p_3}, p_5, \neg p_3$	$\neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9$
$\varnothing$	p_5	$\{p_1, \neg p_1 \vee p_2, \neg p_1 \vee p_3, \neg p_2 \vee \neg p_3\} \vdash \square$
$\varnothing$	$\varnothing$	$\neg p_1 \vee p_4, \neg p_4 \vee p_5, \neg p_5 \vee p_6, \neg p_6 \vee p_7, \neg p_7 \vee \neg p_8, \neg p_7 \vee \neg p_9, p_8 \vee p_9$

3.4.2 Extending lower bound UP with Failed Literal Detection

We can incorporate to lower bound UP* an additional level of forward look-ahead based on the detection of failed literals. We next describe in detail UP^*_{FL}, and assume that inconsistent subformulas in $\phi' \cup \{p\}$ and $\phi' \cup \{\neg p\}$ are detected via UP*. Lower bound UP* has been selected because it is generally better than UP and UP^S, but the previous result holds for UP and UP^S as well (named UP_{FL} and UP^S_{FL} respectively in the experimental results in this chapter).

Let ϕ be a MAX-SAT instance, and let ϕ' be the formula resulting from ϕ after replacing every inconsistent subformula detected by UP* with an empty clause. Obviously, unit propagation in ϕ' cannot derive any additional empty clause. However, if unit propagation is applied to $\phi' \cup \{p\}$ and $\phi' \cup \{\neg p\}$, for any variable p occurring in ϕ', and produces an empty clause in each CNF formula (i.e., p and $\neg p$ are *failed literals* in ϕ'), then $(\varphi_1 \cup \varphi_2) \setminus \{p, \neg p\}$ is an inconsistent subformula of ϕ', where φ_1 is the inconsistent subformula detected by UP* in $\phi' \cup \{p\}$, and φ_2 is the inconsistent subformula detected by UP* in $\phi' \cup \{\neg p\}$. That is a direct consequence of the following observation: We can produce a proof of $\neg p$ by applying resolution to $\varphi_1 \setminus \{p\}$, and a proof of p by applying resolution to $\varphi_2 \setminus \{\neg p\}$. If we put the two proofs together and resolve p and $\neg p$, we get a refutation from $(\varphi_1 \cup \varphi_2) \setminus \{p, \neg p\}$. Note that now: (i) we only consider clauses of ϕ', and (ii) the refutation is a resolution refutation, i.e., it is not restricted to *unit* resolution refutations.

Algorithm 3.6: `FailedLiteral`(ϕ', underestimation) : Computation of lower bound Failed Literal

```
Function FailedLiteral(φ' : CNF formula, underestimation: Natural) : Integer
    forall variable p ∈ Var_FL(φ') do
        φ1 ← UnitPropagation(φ' ∪ {p})
        φ2 ← UnitPropagation(φ' ∪ {¬p})
        if φ1 ≠ ∅ ∧ φ2 ≠ ∅ then
            ▷ Both produce an empty clause ◁
            φ' ← φ' \ (φ1 ∪ φ2 \ {p, ¬p})
            underestimation ← underestimation + 1
    return underestimation
```

Time complexity : $\mathcal{O}(|\phi'| \cdot n)$

Algorithm 3.6 shows the application of failed literal detection after any of the lower bounds UP, UP^S or UP*. Assuming that the time complexity of `UnitPropagation` is $\mathcal{O}(|\phi'|)$, and the fact that the loop has to deal with all the n variables in the worst case, the time complexity of the algorithm is $\mathcal{O}(|\phi'| \cdot n)$.

Example 3.8 *Let ϕ' be $\{p_2 \vee \neg p_1, \neg p_2 \vee p_3, \neg p_2 \vee \neg p_3, p_2 \vee p_1\}$. If unit propagation is applied to $\phi' \cup \{p_1\}$, UP* detects the inconsistent subformula $\varphi_1 = \{p_1, p_2 \vee$*

$\neg p_1, \neg p_2 \vee p_3, \neg p_2 \vee \neg p_3\}$, *and if it is applied to* $\phi' \cup \{\neg p_1\}$, *UP* detects the inconsistent subformula* $\varphi_2 = \{\neg p_1, \neg p_2 \vee p_3, \neg p_2 \vee \neg p_3, p_2 \vee p_1\}$. *Observe that a resolution refutation can be derived from* $(\varphi_1 \cup \varphi_2) \setminus \{p_1, \neg p_1\} = \{p_2 \vee \neg p_1, \neg p_2 \vee p_3, \neg p_2 \vee \neg p_3, p_2 \vee p_1\}$.

As introducing an additional level of look-ahead is time consuming, only a subset of the variables occurring in the CNF formula are used to detect failed literals. Let $Var_{FL}(\phi')$ be the set of propositional variables occurring in ϕ' such that (i) they do not occur in unit clauses; and (ii) they have at least two positive occurrences and two negatives occurrences in binary clauses. UP^*_{FL} detects, for each variable p in $Var_{FL}(\phi')$, if p and $\neg p$ are both failed literals in ϕ'. Once an inconsistent subformula φ is detected, φ is replaced with an empty clause in ϕ', and $Var_{FL}(\phi')$ is updated taking into account the new CNF formula derived.

In the definition of $Var_{FL}(\phi')$, variables occurring in unit clauses are not considered because they did not lead to a contradiction when UP* was applied to ϕ'. Selecting variables with at least two positive occurrences and two negatives occurrences in binary clauses was empirically determined. These variables give at least two new unit clauses when they are set to a truth value.

UP^*_{FL} computes, in general, tighter bounds (the total number of empty clauses in the resulting CNF formula) than UP* and, in the worst-case, it provides the same lower bound as UP*. It is also important to highlight some side effects of its application: (i) as soon as the new lower bound reaches the upper bound for some variable p, we can prune the current search subspace, and (ii) if the difference between the current lower bound and the upper bound is one, and unit propagation in $\phi' \cup \{p\}$ ($\phi' \cup \{\neg p\}$) leads to an empty clause, then p can be set to false (true) (refer to Section 4.1).

3.5 Empirical evaluation

We first define the benchmarks used in the experimental evaluation, and then report on the experimental results.

3.5.1 Benchmarks

In the experimentation, we address two problems: random MAX-k-SAT and random MAX-CUT, a problem that can be reduced to MAX-SAT.

Random MAX-k-SAT

In order to generate random instances, the generation scheme commonly used in the literature during the last years is random k-SAT, proposed by [FP83] and that we introduce here.

A random k-SAT instance is defined according to the three following parameters: number n of variables, number c of clauses, number k of literals per clause.

Given a number n of variables and a number c of clauses, one random k-SAT instance is produced by selecting uniformly, independently, and with replacement c non-tautological clauses of length k among the $2^k\binom{n}{k}$ possible clauses.

Experimentally, given a satisfiable random propositional CNF formula ϕ and a fixed number of variables, the fact of significantly increasing the number of clauses in ϕ will end up inevitably in an unsatisfiable formula. Thus, decreasing the number of clauses will lead to make ϕ satisfiable. This phenomenon, both well studied in practice as in theory, is called *phase transition*.

Since we are interested on unsatisfiable formulas, we show the behavior of a MAX-SAT solver with plots ranging from the phase transition point, where the instances start becoming unsatisfiable, to the saturation point, where instances are forced to have repeated clauses (e.g., saturation point for MAX-2-SAT is $c = 2^2\binom{n}{2} = 2n(n-1)$). In the experimentation, the saturation point was reached whenever possible.

For the creation of random MAX-SAT instances, we have used generator `mwff`, created by Bart Selman and available from DIMACS[3]. It allows the creation of repeated clauses.

MAX-CUT

Let $G = (V, E)$ be an undirected graph. A cut is a partition of the vertices V into two sets S and T. Any edge $(u, v) \in E$ with $u \in S$ and $v \in T$ is said to be crossing the cut and is a cut edge. The size of the cut is the number of edges crossing the cut. A Maximum Cut (MAX-CUT) is then defined as a cut of G of maximum size.

In order to map MAX-CUT to MAX-SAT, we used the encoding in [Yan94]: Given a graph with e edges, we created, for each edge (x_i, x_j), exactly two binary clauses $(p_i \vee p_j)$ and $(\neg p_i \vee \neg p_j)$. If ϕ is the collection of such binary clauses, then the MAX-CUT instance has a cut of k edges if, and only if, the MAX-SAT instance has an assignment under which $e + k$ clauses are satisfied.

Notice that this mapping provides structure to the CNF formula. This changes the solver behavior compared with random MAX-SAT, as we show in the next section.

3.5.2 Experimental results

In this chapter we have introduced three UP based lower bounds: UP, UP* and UP^S; and their respective extensions detecting failed literals, UP$_{FL}$, UP$^*_{FL}$ and UP$^S_{FL}$. In this section, we show the results of testing each of the six lower bounds in the branch and bound MAX-SAT solver MaxSatz (see Chapter 6 for a description of the solver).

Here on, for each experiment we display pairwise plots with running time and number of branches.[4] Displaying branching is a good tool to better understand

[3] Available at `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`.

[4] A branch is computed every time a Boolean variable is going to be instantiated to both values.

a given technique. When a technique is added, we have found four cases to be analyzed:

- If branches and time have been reduced, we have found a good technique.
- If branches have been reduced and time has increased, the source code has to be improved (whenever possible).
- If branches and time have been increased, the technique has to be discarded.
- If branches have been increased and time has reduced, something unexpected happened (e.g., the added technique changes other technique behaviour) and the design has to be checked (unless a bug).

Here on, all the experiments have been performed in a Linux cluster, having all nodes processors AMP Opteron (64bits 2GHz) with 1Mb of memory.

Below, we introduce the experimentation on MAX-k-SAT and MAX-CUT.

MAX-2-SAT and MAX-3-SAT

In the first experiment we have compared the three UP based lower bounds: UP, UP* and UP^S. In Figure 3.6, we see that UP* is the best performing lower bound for a low clause to variable ratio, but when the ratio increases UP* and UP^S get almost overlapped in time (there is a difference of 1.2%).

In the second experiment we have compared the three UP based lower bounds extended with failed literal detection: In Figure 3.7, we see that UP$^*_{FL}$ is the best performing lower bound; and UP$_{FL}$ is the worst scaling, although it is the most effective for a low clause to variable ratio, and better than UP$^S_{FL}$ in almost the whole range.

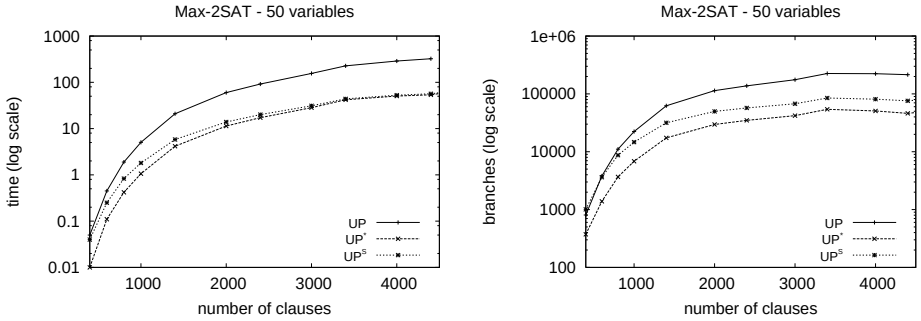


Figure 3.6: Impact of heuristics UP, UP* and UP^S

We have performed a third experiment in order to get insights of the lower bounds performance. One observed effect is the influence of failed literal detection. As can be seen in Figure 3.8, UP^S is not affected as much as the other

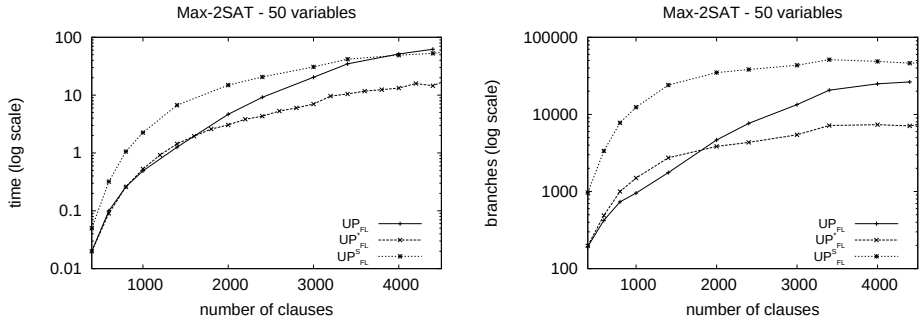


Figure 3.7: Impact of failed literal detection on heuristics UP, UP* and UP^S

two lower bounds. The most probable reason is the number of remaining clauses after the application of the lower bound: there are fewer binary clauses after applying UP^S than after applying UP and UP*. The consequence can be observed in the number of backracks: UP$_{FL}$ and UP$^*_{FL}$ are the ones that consume less backtracks.

In the fourth experiment, we add up the three previous experiments in one plot, Figure 3.9, and observe that UP$^*_{FL}$ is the best performing solver. Surprisingly, UP$_{FL}$ has a good performance in the less constrained instances.

In the fifth experiment, we increased the number of variables to 100. In Figure 3.10 the results show that the best performing solvers are UP$^*_{FL}$, UP* and UP$_{FL}$. In constrast with the previous experiments, UP* plays also a role in the group of the best ones. We observe that failed literal detection has a different behaviour, when observing the number of backtracks, for every lower bound: (i) on lower bound UP^S makes no difference, (ii) on lower bound UP* there is a little gap, and (iii) on lower bound UP there is a big gap. We think that failed literal detection reduce more the number of backtracks in this last case because lower bound UP leaves more inconsistencies to be detected than the other lower bounds.

The last experiments over random MAX-k-SAT are reported in Figure 3.11 and Figure 3.12. We can see the influence of all of the lower bounds for MAX-3-SAT with 50 and 70 variables. We observe the same influence of the heuristics as seen in the previous detailed analysis: UP$^*_{FL}$ is the best performing solver.

MAX-CUT

Finally we report the results for MAX-CUT with 50 nodes. We have performed, like for MAX-2-SAT, a separate experiment for the three UP lower bounds and for their extensions with failed literal detection. In Figure 3.13 we show the lower bounds UP, UP* and UP^S. The best performing is UP*, as seen for random MAX-k-SAT, and the worst performing is UP. UP^S performance goes from UP with low number of edges to UP* when more edges are added. Otherwise, when failed literal detection is added, UP$^S_{FL}$ performs the worst, and UP$_{FL}$ and UP$^*_{FL}$

perform very similar (Figure 3.14). In the right side in the plot, UP_{FL} becomes the best one because it is the lower bound that takes more advantage of failed literal detection, as can be seen in Figure 3.15. The influence of all the lower bounds on MAX-CUT is shown in Figure 3.16.

3.6 Summary

This chapter describes the work on the definition, efficient implementation and analysis of MAX-SAT lower bounds on a branch and bound algorithm. Unit propagation, one of the more useful techniques in SAT, has been extensively used in the computation of efficient lower bounds. In the application of unit propagation, the manner a unit clause is chosen changes the performance of the algorithm. The goal is to take the minimum number of initial unit clauses to detect the maximum number of disjoint inconsistent subsets of clauses. Three different lower bounds have been defined: UP, UP^S and UP^*.

When there is no unit clause to apply unit propagation, or the existing unit clauses do not bring any conflict, an estimation can be made detecting failed literals. This technique puts the solver a step forward. An effect observed in the experimentation is the deterioration of failed literal detection when improving lower bound UP. This is because lower bound UP leaves fewer inconsistencies to be detected.

Many lower bound computations, done in a node in the search tree, have to be computed again in downward nodes. Many times, this re-computation can be avoided applying inference rules, as is introduced in the next chapter.

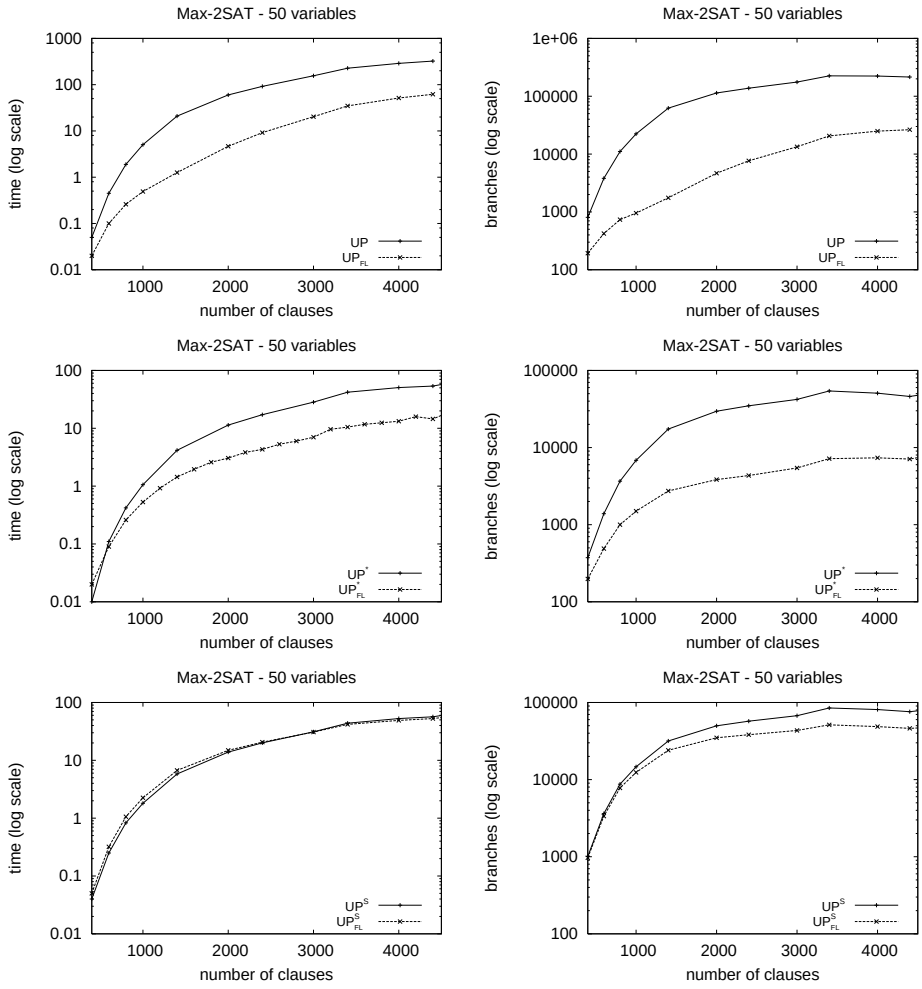


Figure 3.8: Impact of failed literal detection on heuristics UP, UP* and UP^S

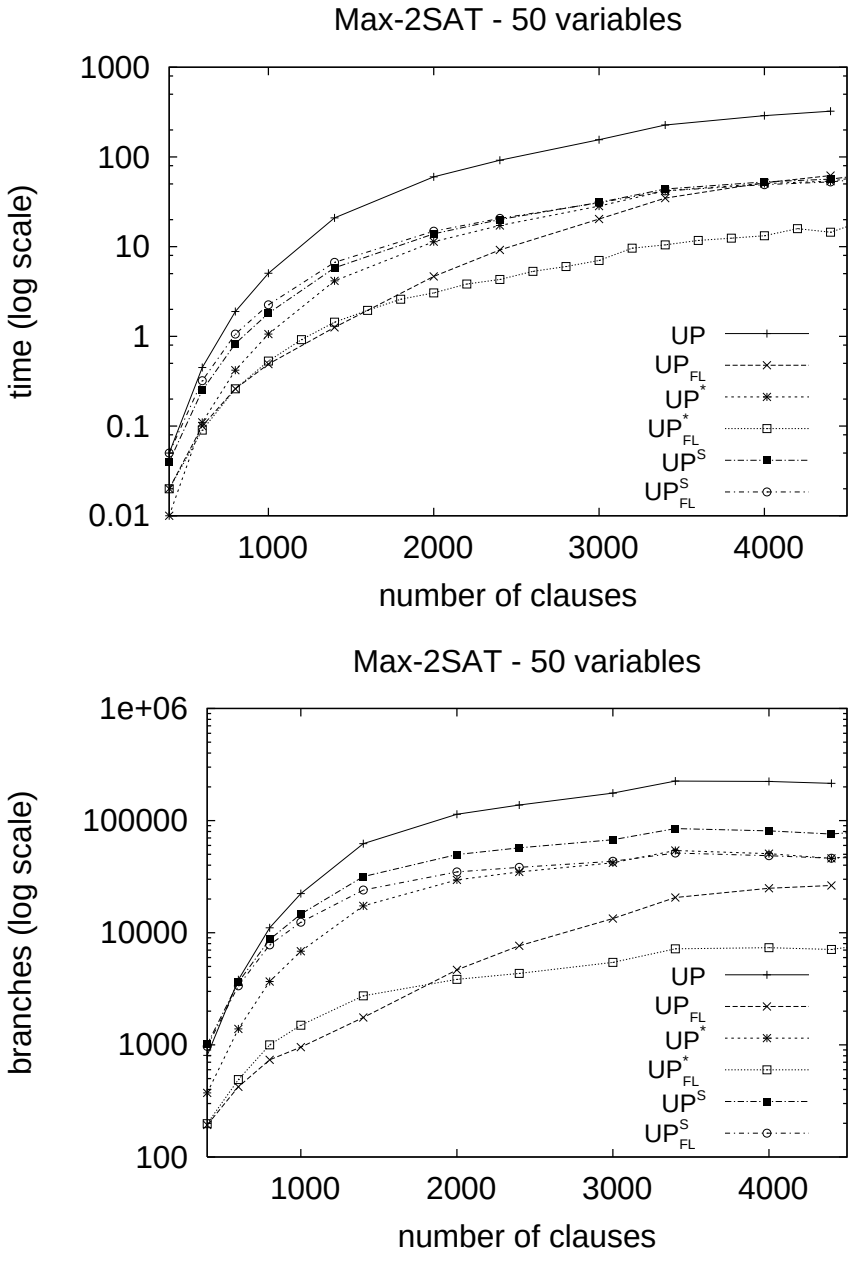


Figure 3.9: Random MAX-2-SAT with 50 variables

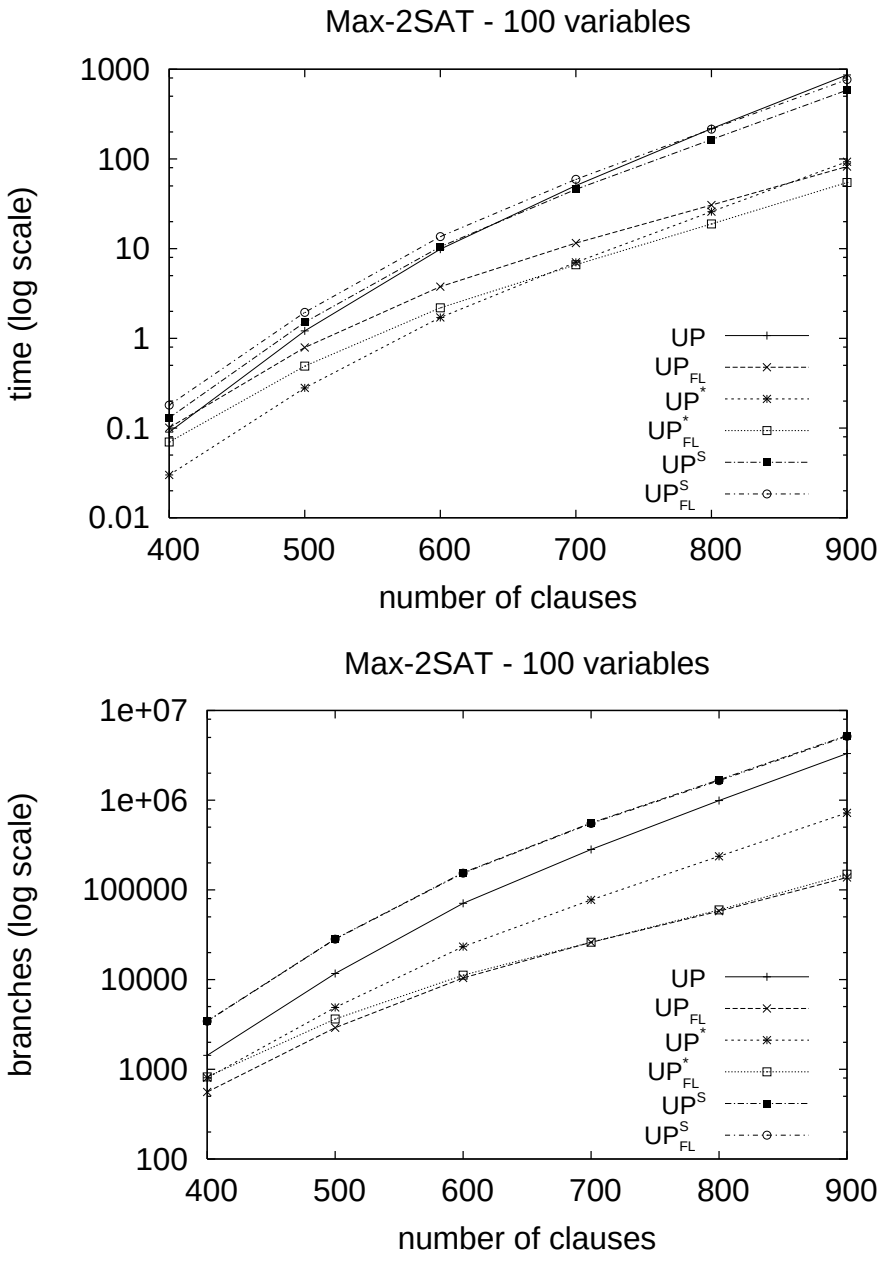


Figure 3.10: Random MAX-2-SAT with 100 variables

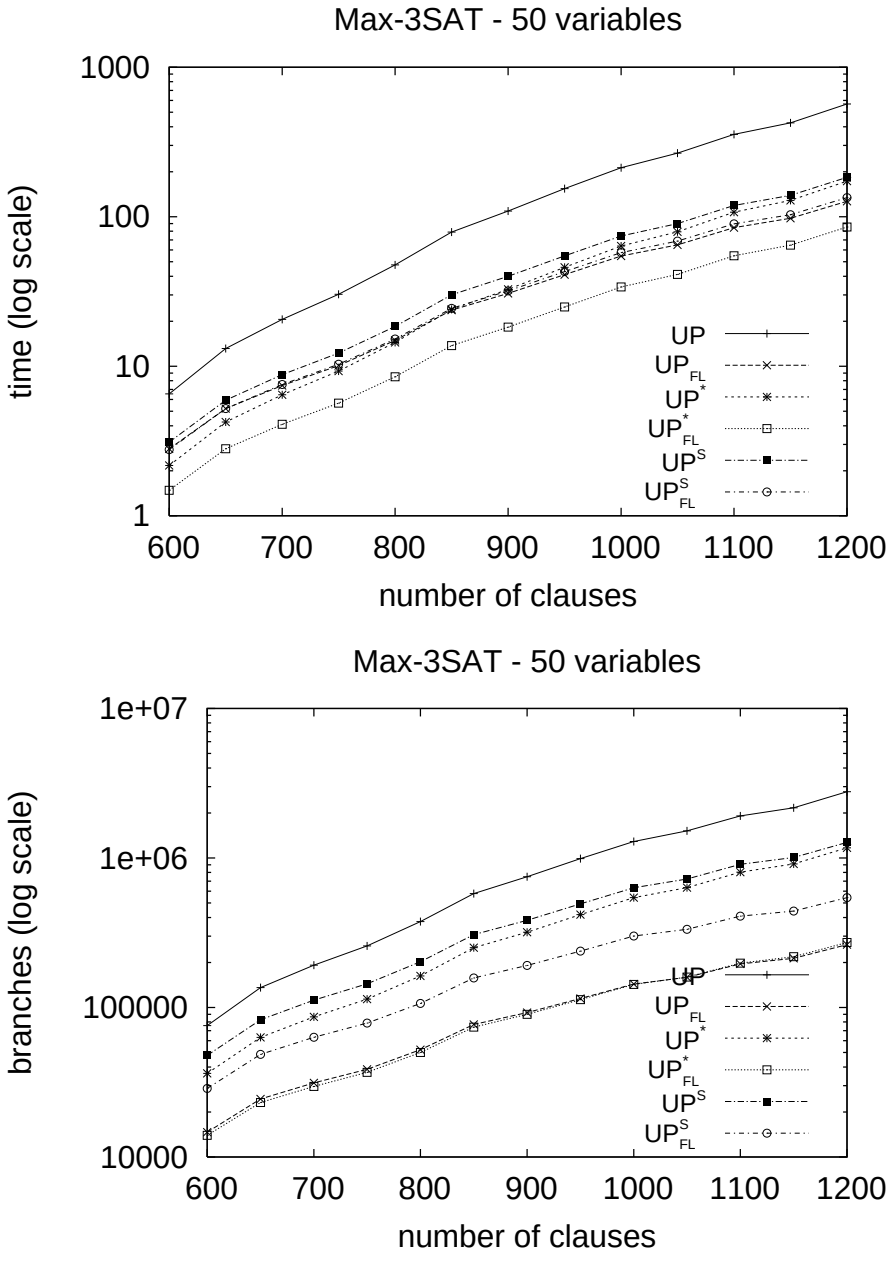


Figure 3.11: Random MAX-3-SAT with 50 variables

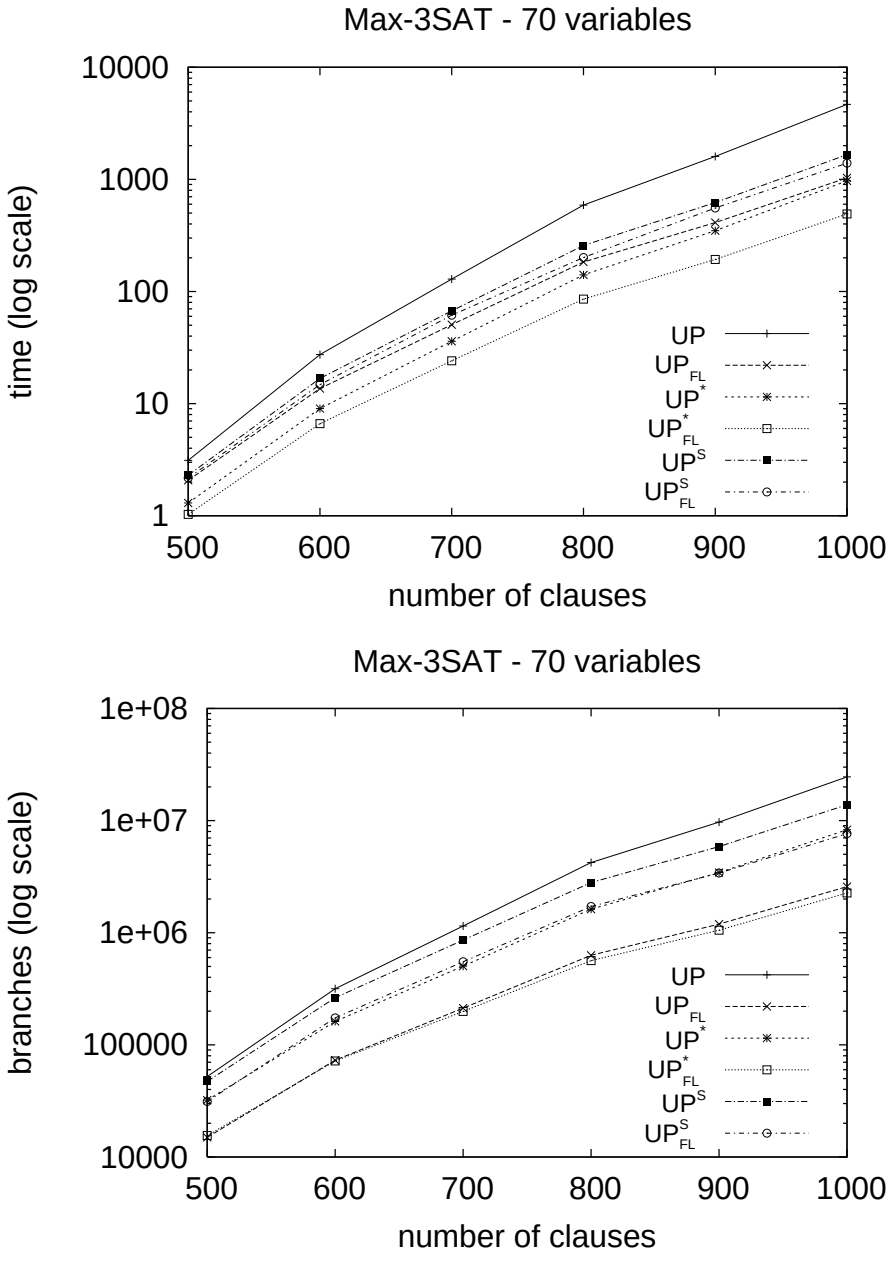


Figure 3.12: Random MAX-3-SAT with 70 variables

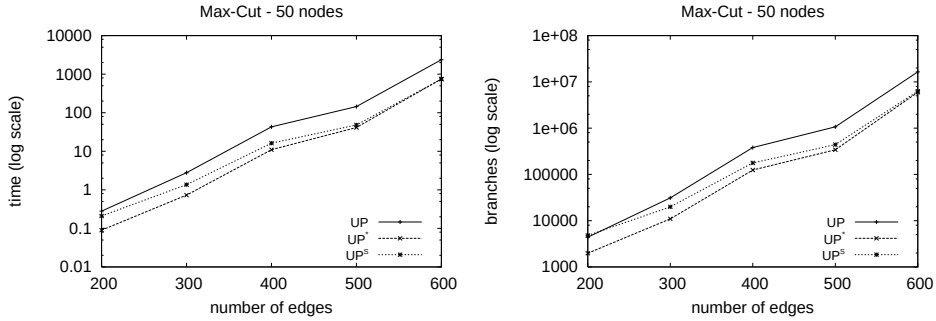


Figure 3.13: Impact of heuristics UP, UP* and UP^S on MAX-CUT

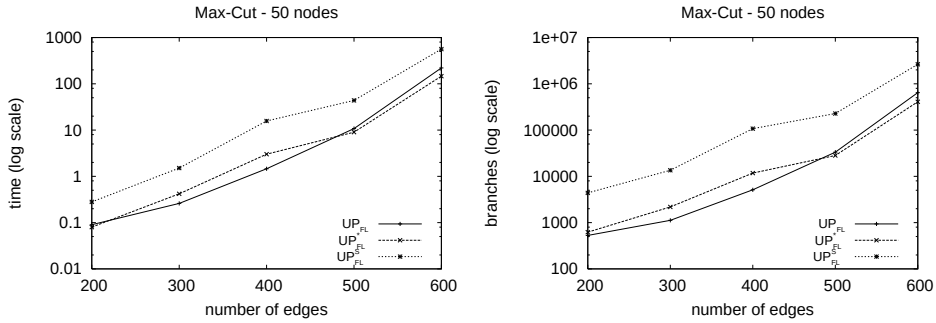


Figure 3.14: Impact of failed literal detection on heuristics UP, UP* and UP^S on MAX-CUT

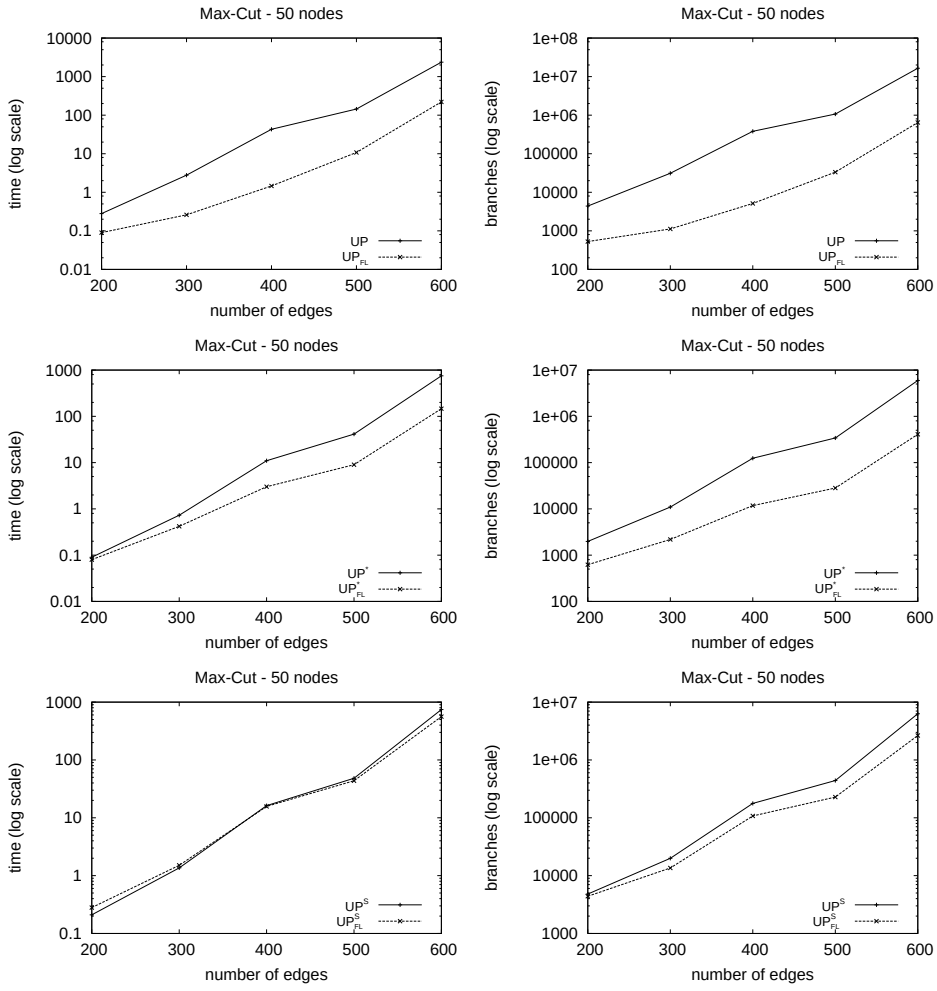


Figure 3.15: Impact of failed literal detection on heuristics UP, UP* and UP^S in MAX-CUT

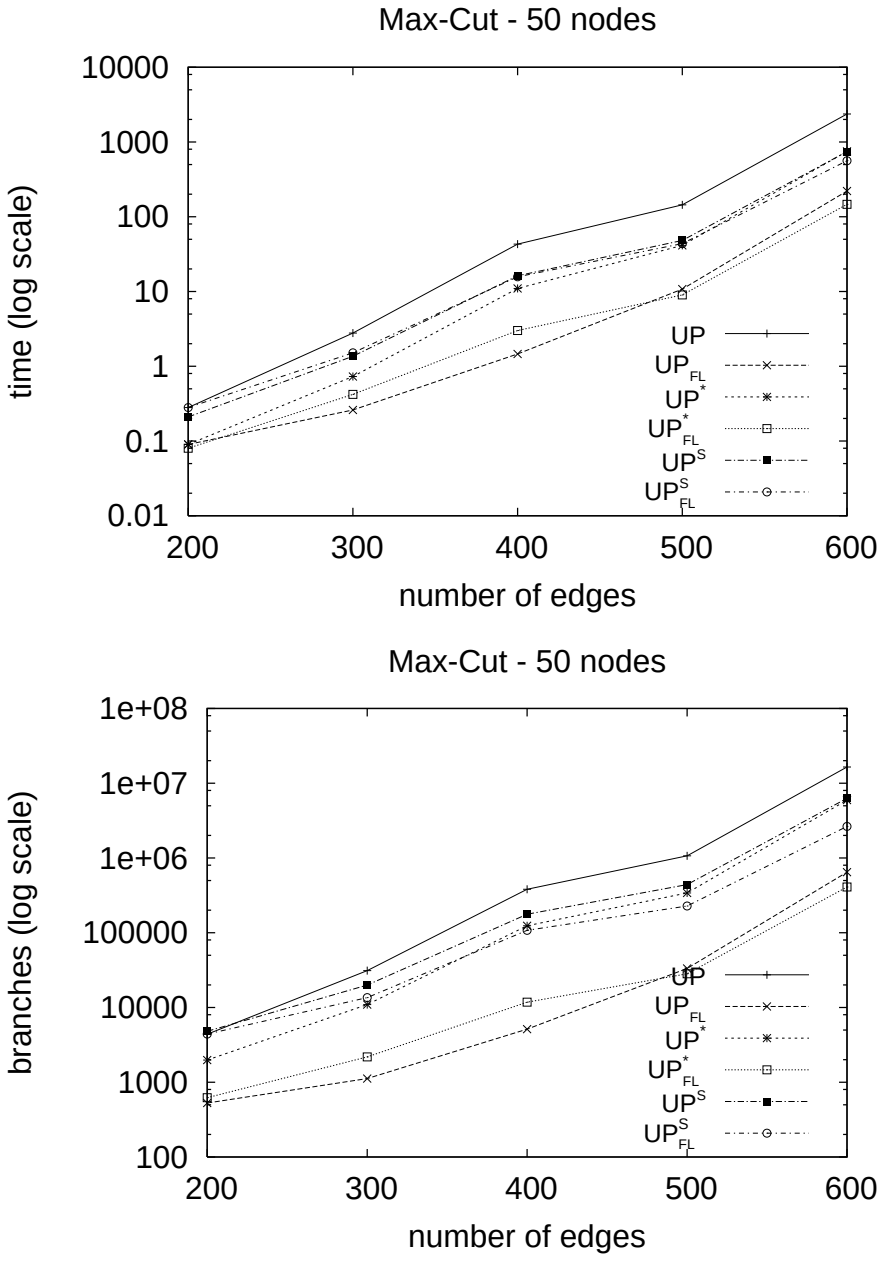


Figure 3.16: Random MAX-CUT with 50 variables

Chapter 4

Inference rules

The amount of inference performed by a branch and bound MAX-SAT solver at each node of the search tree is poor compared with the inference performed in DLL-style SAT solvers. The inference rules that one can apply in MAX-SAT have to transform the current instance ϕ into another instance ϕ' in such a way that ϕ and ϕ' have the same number of unsatisfied clauses for every possible assignment; in other words, the inference rules have to be *sound*. It is not enough to preserve satisfiability as in SAT. Unfortunately, unit propagation, which is the most powerful inference technique applied in SAT, is unsound for MAX-SAT,[1] and many MAX-SAT solvers apply rules which are far from being as powerful as unit propagation in SAT.

The basic MAX-SAT algorithm, when branches on a literal l, enforces the following inference: removes the clauses containing l and deletes the occurrences of $\bar{l}$, but the new unit clauses derived as a consequence of deleting the occurrences of $\bar{l}$ are not propagated as in the DPLL algorithm. Typically, that inference is enhanced by applying simple inference rules such as (i) the pure literal rule [BR99, BF99]; (ii) the dominating unit clause rule [NR00], (iii) the almost common clause rule [BR99], and (iv) the complementary unit clause rule [NR00]. All these rules, which are sound but not complete, have proved to be useful in a number of MAX-SAT solvers [AMP03b, AMP05, BF99, SZ04, XZ05].

In this thesis, we make a step forward by incorporating more sophisticated inference rules, which can be seen as an adaptation to MAX-SAT of some unit resolution refinements. While the rules cited in the previous paragraph can be seen as SAT inference rules that can be safely applied to MAX-SAT, our new rules replace a set of clauses S with a set of clauses S' in such a way that the number of unsatisfied clauses in S and S' is the same for every assignment. S' contains standard resolvents of clauses in S plus some additional clauses that ensure the soundness of the rule. In standard resolution, the conclusion of the inference rule is added to the premises, but this cannot be done in MAX-SAT

[1] The set of clauses $\{p,\ \neg p \vee q,\ \neg p \vee \neg q,\ \neg p \vee r,\ \neg p \vee \neg r\}$ has a minimum of one unsatisfied clause (setting p to false). However, performing unit propagation with p leads to a non-optimal assignment falsifying at least two clauses.

because this could increase the number of unsatisfied clauses.

Let us see an example of one of our resolution-style rules for MAX-SAT:

Given a MAX-SAT instance ϕ that contains three clauses of the form $l_1, l_2, \bar{l}_1 \vee \bar{l}_2$, where l_1, l_2 are literals, replace ϕ with the CNF formula

$$\phi' = (\phi - \{l_1, l_2, \bar{l}_1 \vee \bar{l}_2\}) \cup \{\square, l_1 \vee l_2\}.$$

Note that the rule detects a contradiction from $\{l_1, l_2, \bar{l}_1 \vee \bar{l}_2\}$ and, therefore, replaces these clauses with an empty clause. In addition, the rule adds the clause $l_1 \vee l_2$ to ensure the equivalence between ϕ and ϕ'. For any assignment containing either $l_1 = 0, l_2 = 1$, or $l_1 = 1, l_2 = 0$, or $l_1 = 1, l_2 = 1$, the number of unsatisfied clauses in $\{l_1, l_2, \bar{l}_1 \vee \bar{l}_2\}$ is 1, but for any assignment containing $l_1 = 0, l_2 = 0$, the number of unsatisfied clauses is 2. Since $l_1 \vee l_2$ is unsatisfied for $l_1 = 0, l_2 = 0$, the rule takes into account that situation.

In the rest of the chapter we define a set of unit resolution refinements for MAX-SAT, describe an efficient way of implementing the application of the rules at each node of the search tree, and report on an experimental evaluation that provides empirical evidence that our rules can speed up a MAX-SAT solver several orders of magnitude.

4.1 Related work

As was mentioned in Chapter 2, the first implemented inference rule for MAX-SAT was due to Wallace and Freuder [WF96], who applied a strategy similar to forward checking in CSP [McG79]. Notice that in the Boolean case, reducing the domain means fixing the variables. Using the notation used in the previous chapter, the algorithm of the rule MAX-SAT-FC is:

forall *variable* $p \in Var(\phi)$ **do**
  if $\mathit{EmptyClauses}(\phi) + \min(\mathrm{ic}(p), \mathrm{ic}(\neg p)) + |\mathrm{ic}(p) - \mathrm{ic}(\neg p)| \geq \mathit{UpperBound}(\phi)$ **then**
    if $\mathrm{ic}(p) > \mathrm{ic}(\neg p)$ **then** $\Phi_{\neg p}$ **else** ϕ_p

This inference rule can be improved if the computation of $\texttt{EmptyClauses}(\phi)$ is replaced by a lower bound. This can be done if the clauses involved in the lower bound are not involved in the the inconsistencies detected by the inconsistency count $\mathrm{ic}(\ell)$. This improvement was implemented in the solver in [AMP04a].

A different inference rule is applied in Borchers and Furman's solver [BF99]: when the difference between the number of empty clauses and the upper bound is 1 ($UpperBound(\phi) - EmptyClauses(\phi) = 1$), unit propagation can be safely applied (in fact, a SAT solver can be used). The condition to apply such an inference rule is difficult to happen when the instance solution has many unsatisfied clauses and a good lower bound is computed. This technique was extended to weighted MAX-SAT. In this case the condition to apply the inference rule is $UpperBound(\phi) - EmptyClauses(\phi) = \omega_{min}$, where ω_{min} is the minimum weight of the clauses.

A relevant previous work on inference rules was due to Bansal and Raman [BR99], and Niedermeier et al.[NR00]. They defined a set of rules to be applied to a MAX-SAT formula, that Gramm implemented in an MAX-2-SAT algorithm [GN00]. These inference rules were applied at each node of the proof tree in order to simplify the formula associated to the node. The six rules are as follows:

Pure literal Rule If a variable only appears with either positive polarity or negative polarity, delete the clauses containing that literal.

Complementary Unit Clause (CUC) Rule If a literal appears with both positive and negative polarity in two unit clauses, remove the two clauses and add an empty clause.

Dominating Unit Clause (DUC) Rule This inference rule allows fixing the truth value of a variable. DUC is defined as follows: If the number of clauses (of any length) in which a literal $\bar{\ell}$ appears is not bigger than the number of *unit* clauses in which the literal ℓ appears, then the literal ℓ can be set to true.

(Restricted) Resolution Rule Let ϕ a CNF formula having two clauses of the form $\{p \vee A\}$ and $\{\neg p \vee B\}$, and the rest of clauses of ϕ having no occurrences of variable p, then both clauses are removed and the clause $\{A \vee B\}$ is added to ϕ.

Almost Common Clauses (ACC) Rule If there exist clauses of the form $\{\ell \vee A\}$ and $\{\bar{\ell} \vee A\}$, then replace both clauses with the clause $\{A\}$. This rule was introduced by Bansal and Raman [BR99]. This rule subsumes CUC Rule when $A = \square$.

Three Occurrence Rules They consider two sub-cases:

1. If p occurs 2 times in one polarity and 1 in the other, $\phi = \{\{p \vee q\}, \{p \vee q\}, \{\bar{p} \vee \bar{q}\}\} \cup \phi'$ and ϕ' does not contain any occurrence of variable p, then remove the three clauses.
2. If p occurs 2 times in one polarity and 1 in the other, $\phi = \{\{p \vee q\}, \{p \vee q\}, \{\bar{p} \vee \ell\}\} \cup \phi'$ or $\phi = \{\{p \vee q\}, \{p \vee \ell\}, \{\bar{p} \vee \bar{q}\}\} \cup \phi'$, then replace ϕ with $\{q \vee \ell\} \cup \phi'$ or $\{\bar{q} \vee \ell\} \cup \phi'$, respectively.

Xing and Zhang [XZ05] defined the non-linear integer programming inference rule named UP4. This novel inference rule is based on the integer programming formulation introduced in Chapter 3. Boolean variables p_i in the formula are converted into 0-1 variables x_i, discrete variables taking values 0 or 1. Then, the coefficients for each variable in the obtained integer equation are grouped, so that the following pattern is obtained:

$$c + \sum_{x_i \in V} \pi_i x_i + \sum_{x_i, x_j \in V} \pi_{i,j} x_i x_j + \cdots$$

c is a constant, and $\pi_i, \pi_{i,j}, \pi_{i,j,k}, \ldots$ are the coefficients of items $x_i, x_i x_j, x_i x_j x_k, \ldots$, respectively. The inference rule takes a lower bound $LB(x_i)$ of the coefficients of x_i and fix the value of p_i if the lower bound takes a positive value or zero. It also takes an upper bound $UB(x_i)$ of the coefficients of x_i, and fix the value of p_i if the lower bound takes a negative value or zero. If both conditions hold, the variable can be set to any value.

Example 4.1 *Given ϕ be a CNF formula $\neg p_1, \neg p_2, (p_1 \vee p_2)$, its integer programming formulation is $1 + x_1 + x_2 - x_1 x_2$. Then, the coefficients are $\pi_1 = 1$, $\pi_2 = 1$, and $\pi_{1,2} = -1$. The lower bound for the coefficients of the variables are $LB(x_1) = 0$ and $LB(x_2) = 0$. Then, the variables p_1 and p_2 should be fixed to* false.

Important contributions have its origin in constraint programming. de Givry et al. [dGLMS03] first encoded MAX-SAT as a constraint network and solved it with toolbar [LMS99], a weighted CSP solver that enforces weighted CSP local consistencies (node consistency, arc consistency, directional arc consistency and full directional arc consistency [LS03, LS04]). This solver was later enhanced with existential directional arc consistency [dGZHL05]. Larrosa and Heras also introduced the relation among local consistency in weighted CSP and inference rules in MAX-SAT [LH05a]. They found that the application of CUC, ACC and MAX-SAT-FC (called in the article RES_0, RES_1 and BR3+UCR, respectively) until quiescence enforces arc-consistency AC*. Recently, Heras and Larrosa added new weighted MAX-SAT inference rules [HL06b] to the solver (refer to Section 6.1.1 for a description of the solver). The new rules are:[2]

Directed Resolution If $\phi_1 = \{l_1,\ \bar{l}_1 \vee l_2\} \cup \phi'$ and $\phi_2 = \{l_2,\ \bar{l}_2 \vee l_1\} \cup \phi'$, then ϕ_1 and ϕ_2 are equivalent.

Hyper Resolution There are two cases:

2-RES If $\phi_1 = \{l_1 \vee l_2,\ l_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3\} \cup \phi'$ and $\phi_2 = \{l_1,\ \bar{l}_1 \vee \bar{l}_2 \vee \bar{l}_3,\ l_1 \vee l_2 \vee l_3\} \cup \phi'$, then ϕ_1 and ϕ_2 are equivalent.

3-RES If $\phi_1 = \{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3,\ \bar{l}_3\} \cup \phi'$ and $\phi_2 = \{\square,\ l_1 \vee \bar{l}_2,\ l_2 \vee \bar{l}_3\} \cup \phi'$, then ϕ_1 and ϕ_2 are equivalent.

4.2 UP based inference rules

We define a set of novel inference rules. They were inspired by different unit resolution refinements applied in SAT, and were selected because they could be applied in a natural and efficient way. Some of them are already known in the literature [BR99, NR00], others are original.

Before presenting the rules, we define an integer programming transformation of a CNF formula used to establish the soundness of the rules. The method of proving soundness is novel in MAX-SAT, and provides clear and short proofs.

[2] For the sake of clearness, the rules are introduced in the unweighted version. Applying Property 5.3, the rules can be transformed into the weighted version, as it appears in the article. The notation has also been adapted.

4.2.1 Integer programming transformation of a CNF formula

Assume that $\phi = \{c_1, \ldots, c_m\}$ is a CNF formula with m clauses over the variables $p_1, \ldots, p_n$. Let c_i $(1 \leq i \leq m)$ be $p_{i_1} \vee \cdots \vee p_{i_k} \vee \neg p_{i_{k+1}} \vee \cdots \vee \neg p_{i_{k+r}}$. Note that all positive literals in c_i are put before the negative ones.

We consider all the variables in c_i as integer variables taking values 0 or 1, and define the integer transformation of c_i as

$$\mathcal{E}_i(p_{i_1}, \ldots, p_{i_k}, p_{i_{k+1}}, \ldots, p_{i_{k+r}}) = (1 - x_{i_1}) \cdots (1 - x_{i_k}) x_{i_{k+1}} \cdots x_{i_{k+r}}.$$

Obviously, $\mathcal{E}_i$ has value 0 if, and only if, at least one of the variables x_{i_j}'s $(1 \leq j \leq k)$ is instantiated to 1 or at least one of the variables x_{i_s}'s $(k+1 \leq s \leq k+r)$ is instantiated to 0. In other words, $\mathcal{E}_i$=0 if, and only if, c_i is satisfied. Otherwise $\mathcal{E}_i$=1.

A literal l corresponds to an integer denoted by l itself for our convenience. The intention of the correspondence is that the literal l is satisfied if the integer l is 1, and is unsatisfied if the integer l is 0. So, if l is a positive literal x, the corresponding integer l is x, $\bar{l}$ is $1 - x = 1 - l$, and if l is a negative literal $\bar{x}$, l is $1 - x$ and $\bar{l}$ is $x = 1 - (1 - x) = 1 - l$. Consequently, $\bar{l} = 1 - l$ in any case.

We now generically write c_i as $l_1 \vee l_2 \vee \cdots \vee l_{k+r}$. The integer programming transformation of c_i is

$$\mathcal{E}_i = (1 - l_1)(1 - l_2) \cdots (1 - l_{k+r}).$$

The integer programming transformation of a CNF formula $\phi = \{c_1, \ldots, c_m\}$ over the variables $x_1, \ldots, x_n$ is defined as

$$\mathcal{E}(x_1, \ldots, x_n) = \sum_{i=1}^{m} \mathcal{E}_i \tag{4.1}$$

That integer programming transformation was used in [HC97, LH05b] to design a local search procedure. Here, we extend it to empty clauses: if c_i is empty, then $\mathcal{E}_i$=1.

Given an assignment A, the value of $\mathcal{E}$ is the number of unsatisfied clauses in ϕ. If A satisfies all clauses in ϕ, then $\mathcal{E} = 0$. Obviously, the minimum number of unsatisfied clauses of ϕ is the minimum value of $\mathcal{E}$.

Let ϕ_1 and ϕ_2 be two CNF formulas, and let $\mathcal{E}_1$ and $\mathcal{E}_2$ be their integer programming transformations. It is clear that solving the MAX-SAT problem for ϕ_1 is equivalent to solving it for ϕ_2 if, and only if, $\mathcal{E}_1 = \mathcal{E}_2$. In the sequel, when we say that ϕ_1 and ϕ_2 are equivalent, we mean that solving the MAX-SAT problem for ϕ_1 is equivalent to solving it for ϕ_2.

4.2.2 Inference rules

We next define the inference rules and prove their soundness using the previous integer programming transformation.[3] In the rest of the section, ϕ_1, ϕ_2 and ϕ'

[3]It is worth to mention that the rules can be also demonstrated using the resolution rule introduced in [BLM06].

denote CNF formulas, and $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}'$ their integer programming transformations. To prove that ϕ_1 and ϕ_2 are equivalent, we prove that $\mathcal{E}_1 = \mathcal{E}_2$.

Rule 4.1 (ACC) *[BR99] If* ϕ_1=$\{l_1 \vee l_2 \vee \cdots \vee l_k,\ \bar{l}_1 \vee l_2 \vee \cdots \vee l_k\} \cup \phi'$ *and* ϕ_2=$\{l_2 \vee \cdots \vee l_k\} \cup \phi'$*, then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof

$$\begin{aligned} \mathcal{E}_1 &= (1-l_1)(1-l_2)\cdots(1-l_k) + l_1(1-l_2)\cdots(1-l_k) + \mathcal{E}' \\ &= (1-l_2)\cdots(1-l_k) + \mathcal{E}' \\ &= \mathcal{E}_2 \quad \blacksquare \end{aligned}$$

Rule 4.1 is known in the literature as *replacement of almost common clauses* (cf. Section 4.1). We pay special attention to the case k=2, where the resolvent is a unit clause, and to the case k=1, where the resolvent is the empty clause. We describe this latter case in the following rule:

Rule 4.2 (CUC) *[NR00] If* ϕ_1=$\{l,\ \bar{l}\} \cup \phi'$*,* ϕ_2=$\{\square\} \cup \phi'$*, then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof $\mathcal{E}_1 = 1 - l + l + \mathcal{E}' = 1 + \mathcal{E}' = \mathcal{E}_2$ $\blacksquare$

Rule 4.2, which is known as *complementary unit clause rule* [NR00], can be used to replace two complementary unit clauses with an empty clause. The new empty clause contributes to the lower bounds of the search space below the current node by incrementing the number of unsatisfied clauses, but not by incrementing the underestimation. Therefore, this contradiction has not to be detected again. In practice, that simple rule gives rise to considerable gains.

Before presenting the following rules, we define a lemma needed to prove their soundness.

Lemma 4.1 *If* ϕ_1=$\{l_1,\ \bar{l}_1 \vee l_2\} \cup \phi'$ *and* ϕ_2=$\{l_2,\ \bar{l}_2 \vee l_1\} \cup \phi'$*, then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof

$$\begin{aligned} \mathcal{E}_1 &= 1 - l_1 + l_1(1-l_2) + \mathcal{E}' \\ &= 1 - l_1 + l_1 - l_1 l_2 + \mathcal{E}' \\ &= 1 - l_2 + l_2 - l_1 l_2 + \mathcal{E}' \\ &= 1 - l_2 + (1-l_1) l_2 + \mathcal{E}' \\ &= \mathcal{E}_2 \quad \blacksquare \end{aligned}$$

It is worth to mention that the application of Lemma 4.1 two times in a formula ϕ brings ϕ again.

The following rule, which is original, is a more complicated case:

Rule 4.3 *If* ϕ_1=$\{l_1,\ \bar{l}_1 \vee \bar{l}_2,\ l_2\} \cup \phi'$ *and* ϕ_2=$\{\square,\ l_1 \vee l_2\} \cup \phi'$*, then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof Applying Lemma 4.1 to the first two clauses of ϕ_1, we get $\phi'_1 = \{l_1 \vee l_2,\ \bar{l}_2,\ l_2\}$, and applying Rule 4.2 to the last two clauses of ϕ'_1, we get $\{\square, l_1 \vee l_2\}$. ■

Rule 4.3 replaces three clauses with an empty clause, and adds a new binary clause to keep the equivalence between ϕ_1 and ϕ_2.

That pattern was considered to compute a lower bound in [AMP04a, SZ04], and is also captured by our method of computing underestimations based on unit propagation (cf. Chapter 3) .

Let us define a rule that generalizes Rule 4.2 and Rule 4.3.

Rule 4.4 *If* ϕ_1=$\{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3,\ \ldots,\ \bar{l}_k \vee l_{k+1},\ \bar{l}_{k+1}\} \cup \phi'$, ϕ_2=$\{\square,\ l_1 \vee \bar{l}_2,\ l_2 \vee \bar{l}_3,\ \ldots,\ l_k \vee \bar{l}_{k+1}\} \cup \phi'$, *then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof We prove the soundness of the rule by induction on k. When k=1, $\phi_1 = \{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2\} \cup \phi'$. By applying Rule 4.3, we get $\{\square,\ l_1 \vee \bar{l}_2\} \cup \phi'$, which is ϕ_2 when $k = 1$. Therefore, ϕ_1 and ϕ_2 are equivalent.

Assume that Rule 4.4 is sound for $k = n$. Let us prove that it is sound for $k = n + 1$. In that case:

$$\phi_1 = \{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3,\ \ldots,\ \bar{l}_n \vee l_{n+1},\ \bar{l}_{n+1} \vee l_{n+2},\ \bar{l}_{n+2}\} \cup \phi'.$$

By applying Lemma 4.1 to the last two clauses of ϕ_1 (before ϕ'), we get

$$\{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3,\ \ldots,\ \bar{l}_n \vee l_{n+1},\ \bar{l}_{n+1},\ l_{n+1} \vee \bar{l}_{n+2}\} \cup \phi'.$$

By applying the induction hypothesis to the first $n + 1$ clauses of the previous CNF formula, we get

$$\{\square,\ l_1 \vee \bar{l}_2,\ l_2 \vee \bar{l}_3,\ \ldots,\ l_n \vee \bar{l}_{n+1},\ l_{n+1} \vee \bar{l}_{n+2}\} \cup \phi',$$

which is ϕ_2 when $k = n+1$. Therefore, ϕ_1 and ϕ_2 are equivalent and the rule is sound. ■

Rule 4.4 is an original inference rule. It captures linear unit resolution refutations in which clauses and resolvents are used exactly once. The rule simply eliminates the unit and binary clauses used in the refutation, and adds an empty clause and k new binary clauses that are obtained by negating the literals of the eliminated binary clauses. So, all the operations involved can be performed efficiently.

The lower bounds based on unit propagation described in the previous chapter and lower bound LB4 of Shen and Zhang [SZ04] are the only lower bounds, to the best of our knowledge, that capture that kind of inconsistencies. The originality of Rule 4.4 is that it allows to simplify the formula from that inconsistencies.

Note that the added binary clauses may contribute to detect other inconsistencies. They can be used by the procedure that applies the inference rules, as well as by the lower bound computation method.

Finally, we present two new rules that capture unit resolutions refutations in which there is a linear derivation but the unit clause is used twice in the derivation of the empty clause.

Rule 4.5 *If* ϕ_1=$\{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3\} \cup \phi'$ *and* ϕ_2=$\{\Box,\ l_1 \vee \bar{l}_2 \vee \bar{l}_3,\ \bar{l}_1 \vee l_2 \vee l_3\} \cup \phi'$*, then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof

$$\begin{aligned}
\mathcal{E}_1 &= 1 - l_1 + l_1(1-l_2) + l_1(1-l_3) + l_2 l_3 + \mathcal{E}' \\
&= 1 - l_1 + l_1 - l_1 l_2 + l_1 - l_1 l_3 + l_2 l_3 + \mathcal{E}' \\
&= 1 + l_2 l_3 - l_1 l_2 l_3 + l_1 - l_1 l_2 - l_1 l_3 + l_1 l_2 l_3 + \mathcal{E}' \\
&= 1 + (1-l_1) l_2 l_3 + l_1(1 - l_2 - l_3 + l_2 l_3) + \mathcal{E}' \\
&= 1 + (1-l_1) l_2 l_3 + l_1(1-l_2)(1-l_3) + \mathcal{E}' \\
&= \mathcal{E}_2 \quad \blacksquare
\end{aligned}$$

We can combine a linear derivation with Rule 4.5 to obtain Rule 4.6:

Rule 4.6 *If* ϕ_1=$\{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3,\ \ldots,\ \bar{l}_k \vee l_{k+1},\ \bar{l}_{k+1} \vee l_{k+2}, \bar{l}_{k+1} \vee l_{k+3},\ \bar{l}_{k+2} \vee \bar{l}_{k+3}\} \cup \phi'$ *and* ϕ_2=$\{\Box,\ l_1 \vee \bar{l}_2,\ l_2 \vee \bar{l}_3,\ \ldots,\ l_k \vee \bar{l}_{k+1},\ l_{k+1} \vee \bar{l}_{k+2} \vee \bar{l}_{k+3},\ \bar{l}_{k+1} \vee l_{k+2} \vee l_{k+3}\} \cup \phi'$*, then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof We prove the soundness of the rule by induction on k. When k=1,

$$\phi_1 = \{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3, \bar{l}_2 \vee l_4,\ \bar{l}_3 \vee \bar{l}_4\} \cup \phi'.$$

By Lemma 1, we get

$$\{l_1 \vee \bar{l}_2,\ l_2,\ \bar{l}_2 \vee l_3,\ \bar{l}_2 \vee l_4,\ \bar{l}_3 \vee \bar{l}_4\} \cup \phi'.$$

By Rule 4.5, we get

$$\{l_1 \vee \bar{l}_2,\ \Box,\ l_2 \vee \bar{l}_3 \vee \bar{l}_4,\ \bar{l}_2 \vee l_3 \vee l_4\} \cup \phi',$$

which is ϕ_2 when $k = 1$. Therefore, ϕ_1 and ϕ_2 are equivalent.

Assume that Rule 4.6 is sound for $k = n$. Let us prove that it is sound for $k = n + 1$. In that case:

$$\phi_1 = \{l_1,\ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3,\ \ldots,\ \bar{l}_{n+1} \vee l_{n+2},\ \bar{l}_{n+2} \vee l_{n+3},\ \bar{l}_{n+2} \vee l_{n+4},\ \bar{l}_{n+3} \vee \bar{l}_{n+4}\} \cup \phi'.$$

By Lemma 4.1, we get

$$\{l_1 \vee \bar{l}_2,\ l_2,\ \bar{l}_2 \vee l_3,\ \ldots,\ \bar{l}_{n+1} \vee l_{n+2},\ \bar{l}_{n+2} \vee l_{n+3},\ \bar{l}_{n+2} \vee l_{n+4},\ \bar{l}_{n+3} \vee \bar{l}_{n+4}\} \cup \phi'.$$

By applying the induction hypothesis, we get

$$\{l_1 \vee \bar{l}_2,\ \Box,\ l_2 \vee \bar{l}_3,\ \ldots,\ l_{n+1} \vee \bar{l}_{n+2},\ l_{n+2} \vee \bar{l}_{n+3} \vee \bar{l}_{n+4},\ \bar{l}_{n+2} \vee l_{n+3} \vee l_{n+4}\} \cup \phi',$$

which is ϕ_2 when $k = n + 1$. Therefore, ϕ_1 and ϕ_2 are equivalent and the rule is sound. $\blacksquare$

Our lower bound based on unit propagation [LMP05] is the only lower bound that captures the kind of inconsistencies detected by Rule 4.5 and Rule 4.6. Observe that the application of the last two rules can be performed efficiently too.

4.3 On implementing the inference rules

The solver constructs, for each literal, a list with the clauses containing that literal. These lists are constructed when the CNF formula is loaded, and are not incremented or decremented anymore. This makes the algorithm to be efficiently implemented because there is no dynamic memory call during the search.

Rules 4.1 and 4.2 can be efficiently implemented by applying a matching algorithm (refer to [CLRS01] for efficient implementation) over the lists of clauses. Both rules have a time complexity of $\mathcal{O}(|\phi|)$, being $|\phi|$ the length of the CNF formula. These rules are applied at every node, before any lower bound computation or inference rule application.

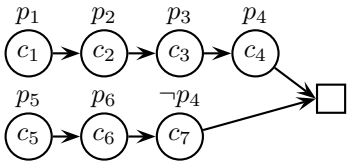
If the lower bound UP detects a contradiction, i.e., if the implication graph G contains both ℓ and $\bar{\ell}$ for some literal ℓ, let S_ℓ be the set of all nodes from which there exists a path to ℓ, $S_{\bar{\ell}}$ be the set of all nodes from which there exists a path to $\bar{\ell}$, and $S{=}S_\ell \cup S_{\bar{\ell}}$. As a clause is associated with each node in G, we also use S, S_ℓ, and $S_{\bar{\ell}}$ to denote the corresponding set of clauses. Lemmas 4.2 and 4.3 are used to detect the applicability of Rules 4.3, 4.4, 4.5, and 4.6.

Lemma 4.2 *Rules 4.3 and 4.4 are applicable if*

1. *there is one unit clause in S_ℓ (resp. $S_{\bar{\ell}}$) and all other clauses are binary,*
2. *nodes in S_ℓ (resp. $S_{\bar{\ell}}$) form an implication chain starting at the unit clause, and ending by ℓ (resp. $\bar{\ell}$),*
3. *$S_\ell \cap S_{\bar{\ell}}$ is empty.*

Proof Starting at the node corresponding to the unit clause in S_ℓ (resp. $S_{\bar{\ell}}$), and following in parallel the two implication chains, we have ϕ_1 in Rule 4.3 or 4.4 by writing down the clause corresponding to each node. ■

Example 4.2 *Let ϕ be the following CNF formula containing clauses c_1 to c_7: $\{c_1 : p_1,\ \ c_2 : \neg p_1 \vee p_2,\ \ c_3 : \neg p_2 \vee p_3,\ \ c_4 : \neg p_3 \vee p_4,\ \ c_5 : p_5,\ \ c_6 : \neg p_5 \vee p_6,\ \ c_7 : \neg p_6 \vee \neg p_4\}$. The two complementary literals in the implication graph G are p_4 and $\neg p_4$. G is as follows:*



Rule 4.4 is applicable, since $\ell{=}p_4$, $S_\ell{=}\{p_1(c_1),\ \ p_2(c_2),\ \ p_3(c_3),\ \ p_4(c_4)\}$, and $S_{\bar{\ell}}{=}\{p_5(c_5),\ \ p_6(c_6),\ \ \neg p_4(c_7)\}$. It is easy to verify that the three conditions of Lemma 4.2 are satisfied.

Remark: ϕ can be rewritten as $\{c_1 : p_1,\ \ c_2 : \neg p_1 \vee p_2,\ \ c_3 : \neg p_2 \vee p_3,\ \ c_4 : \neg p_3 \vee p_4,\ \ c_7 : \neg p_4 \vee \neg p_6,\ \ c_6 : p_6 \vee \neg p_5,\ \ c_5 : p_5\}$ in order to be compared with ϕ_1 in Rule 4.4.

The application of Rules 4.3 and 4.4 consists in replacing each binary clause c in S with a binary clause obtained by negating every literal of c, removing the two unit clauses of S from ϕ, and incrementing $EmptyClauses(\phi)$ by 1.

Lemma 4.3 *Rules 4.5 and 4.6 are applicable if*

1. *there is one unit clause in S=$S_\ell \cup S_{\bar{\ell}}$, and all the other clauses are binary, i.e., all nodes in S have exactly one incoming edge in G, except the node corresponding to the unit clause,*
2. *$S_\ell \cap S_{\bar{\ell}}$ is not empty and contains k ($k>0$) nodes forming an implication chain like $\ell_1 \rightarrow \ell_2 \rightarrow \ ... \ \rightarrow \ell_k$, ℓ_k being the last node of this chain,*
3. *$(S_\ell \cup S_{\bar{\ell}}) - (S_\ell \cap S_{\bar{\ell}})$ exactly contains three nodes : ℓ, $\bar{\ell}$, and a third one. Let ℓ_{k+1} be this third literal,*

   if $\ell_{k+1} \in S_\ell$, then G contains the following implications

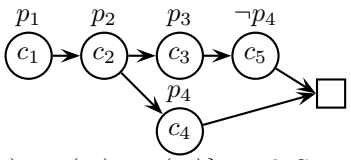
   $$\ell_k \rightarrow \ell_{k+1} \rightarrow \ell$$
   $$\ell_k \rightarrow \bar{\ell}$$

   if $\ell_{k+1} \in S_{\bar{\ell}}$, then G contains the following implications

   $$\ell_k \rightarrow \ell$$
   $$\ell_k \rightarrow \ell_{k+1} \rightarrow \bar{\ell}$$

Proof Without loss of generality, assume $\ell_{k+1} \in S_\ell$; the case $\ell_{k+1} \in S_{\bar{\ell}}$ is symmetric. The implication chain formed by nodes of $S_\ell \cap S_{\bar{\ell}}$ correspond to clauses $\{\ell_1, \bar{\ell}_1 \vee \ell_2, ..., \bar{\ell}_{k-1} \vee \ell_k\}$, which, together with the three clauses $\{\bar{\ell}_k \vee \ell_{k+1}, \bar{\ell}_{k+1} \vee \ell, \bar{\ell}_k \vee \bar{\ell}\}$ corresponding to $\ell_k \rightarrow \ell_{k+1} \rightarrow \ell$ and $\ell_k \rightarrow \bar{\ell}$, give ϕ_1 in Rule 4.5 or Rule 4.6. ■

Example 4.3 *Let ϕ be the following CNF formula containing clauses c_1 to c_5: $\{c_1 : p_1, \quad c_2 : \neg p_1 \vee p_2, \quad c_3 : \neg p_2 \vee p_3, \quad c_4 : \neg p_2 \vee p_4, \quad c_5 : \neg p_3 \vee \neg p_4\}$. unit propagation constructs G with two complementary literals p_4 and $\neg p_4$ as follows:*



We have S_{p_4}=$\{p_1(c_1), \ p_2(c_2), \ p_4(c_4)\}$ and $S_{\neg p_4}$=$\{p_1(c_1), \ p_2(c_2), \ p_3(c_3), \ \neg p_4(c_5)\}$. The nodes in $S_{p_4} \cap S_{\neg p_4}$ form an implication chain: $p_1 \rightarrow p_2$. $(S_{p_4} \cup S_{\neg p_4}) - (S_{p_4} \cap S_{\neg p_4})$=$\{p_3(c_3), \ p_4(c_4), \ \neg p_4(c_5)\}$. G contains $p_2 \rightarrow p_3 \rightarrow \neg p_4$ and $p_2 \rightarrow p_4$. Rule 4.6 is applicable.

The application of Rule 4.5 and Rule 4.6 consists in removing the unit clause of $S_\ell \cup S_{\bar{\ell}}$ from ϕ, replacing each binary clause c in $S_\ell \cap S_{\bar{\ell}}$ with a binary clause obtained from c by negating the two literals of c, replacing the three binary clauses in $(S_\ell \cup (S_{\bar{\ell}})$-$(S_\ell \cap S_{\bar{\ell}})$ by two ternary clauses, and incrementing $EmptyClauses(\phi)$ by 1.

4.3.1 Complexity, termination, and (in)completeness of the applications of the rules

We combine the application of the inference rules and any of the unit propagation based lower bounds in the branch and bound algorithm for MAX-SAT. From here on, we call *underestimation* the function to compute any of such lower bounds. Given a CNF formula ϕ, function *underestimation* uses unit propagation to construct an implication graph G. Once G contains two nodes ℓ and $\bar{\ell}$ for some literal ℓ, G is analyzed to see if there is an applicable inference rule. If so, the rule is applied and ϕ is transformed. Otherwise, all the clauses contributing to the contradiction are removed from ϕ, and the underestimation of unsatisfied clauses in ϕ is increased by 1. This procedure is repeated until unit propagation derives no more contradictions. Finally all the removed clauses, except those removed or replaced by inference rule applications, are inserted into ϕ. The underestimation, together with the new ϕ, is returned by the function (see Section 3.3 for details of the lower bound).

It is well known that unit propagation has linear time complexity in the size of ϕ [GEI91, Fre95]. The detection of applicability of the inference rules using Lemma 4.2 and Lemma 4.3 is linear in the size of G, bounded by the number of literals in ϕ. The application of an inference rule is obviously linear in the size of G. So, the whole time complexity of the *underestimation* function with inference rule applications is in $O(k \cdot |\phi|)$, where k is the number of contradictions that the function can find by unit propagation (see also Section 3.3.2). Notice that the larger the parameter k, the more powerful the function *underestimation* with inference rule applications, in the reduction of the search tree size.

Since every inference rule application reduces the size of ϕ, the *underestimation* function with inference rule applications has linear space complexity. Recall that new clauses in each inference rule selected in our approach can be stored in the place of the old ones, and the data structures for loading ϕ can be statically and efficiently managed.

We have proved that the rules introduced in Section 4.2.2 are sound. The following example shows that the application of the rules is not complete in our implementation.

Example 4.4 *Let* ϕ=$\{p_1, p_3, p_4, \neg p_1 \vee \neg p_3 \vee \neg p_4, \neg p_1 \vee \neg p_2, p_2\}$, *unit propagation called by the* underestimation *function may discover the inconsistent set* S=$\{p_1, p_3, p_4, \neg p_1 \vee \neg p_3 \vee \neg p_4\}$. *In this case, no inference rule is applicable to* S, *which is then removed from* ϕ *to increase by 1 the underestimation of the number of unsatisfied clauses in* ϕ. *Then,* ϕ *becomes* $\{\neg p_1 \vee \neg p_2, p_2\}$. *Unit propagation finds no more contradictions in* ϕ, *and* underestimation *stops after re-inserting* S=$\{p_1, p_3, p_4, \neg p_1 \vee \neg p_3 \vee \neg p_4\}$ *into* ϕ. *The value 1 is returned, together with the unchanged* ϕ. *Note that Rule 4.3 is applicable to the subset* $\{p_1, \neg p_1 \vee \neg p_2, p_2\}$ *of* ϕ *but is not applied.*

Actually, the underestimation *function applies Rule 4.3 only if unit propagation finds the inconsistent subset* $\{p_1, \neg p_1 \vee \neg p_2, p_2\}$ *instead of* $\{p_1, p_3, p_4, \neg p_1 \vee \neg p_3 \vee \neg p_4\}$, *which depends on the ordering of unit clauses used in unit propaga-*

tion. In this example, the inconsistent subset $\{p_1, \neg p_1 \vee \neg p_2, p_2\}$ *is discovered if unit clause* p_2 *is propagated before* p_3 *and* p_4*.*

4.4 Experimental results

In order to compare the six inference rules defined, we have used the solver MaxSatz (see Chapter 6 for a description of the solver) with lower bound UP* (refer to the previous chapter). Then, three solvers have been created, each solver having an increasing number of inference rules:

- `Bare`: does not apply any inference rule.
- `Basic`: applies rules 4.1 and 4.2, but not rules 4.3, 4.4, 4.5 and 4.6.
- `Star`: applies rules 4.1, 4.2, 4.3 and 4.4, but not rules 4.5 and 4.6.
- `All`: applies all the rules.

The experimentation has been performed with the same benchmarks used in the previous chapter: random MAX-k-SAT and random MAX-CUT. Six experiments have been performed: MAX-2-SAT with 50 variables (Figure 4.1) and with 100 variables (Figure 4.2); MAX-3-SAT with 50 variables (Figure 4.3) and 70 variables (Figure 4.4); and MAX-CUT with 50 nodes (Figure 4.5).

We observe that the rules are very powerful for MAX-2-SAT and the gain increases as the number of variables and the number of clauses increase. For 50 variables and 1000 clauses (the clause to variable ratio is 20), `All` is 7.6 times faster than `Star`; for 4000 clauses (the clause to variable ratio is 80), `All` is 19 times faster than `Star`; and for 100 variables and 1000 clauses (the clause to variable ratio is 10), `All` is 9.2 times faster than `Star`. The search tree of `All` is also substantially smaller than the one of `Star`. Rule 4.5 and Rule 4.6 are more powerful than Rule 4.3 and Rule 4.4 for MAX-2-SAT. The intuitive explanation is that `All` and `Star` detect many more inconsistent subsets of clauses containing one unit clause than subsets containing two unit clauses, so that Rule 4.5 and Rule 4.6 can be applied many times more than Rule 4.3 and Rule 4.4 in `All`.

We have shown the behavior of the several rules without failed literal detection. In the next experiments we show the combination of both techniques. In Figure 4.6 and Figure 4.7 we show the results for random MAX-2-SAT with 50 and 100 variables. In MAX-2-SAT, when failed literal detection is added to the rules, `Basic` can take more advantage both reducing the number of branches and the running time.

In Figure 4.8 and Figure 4.9 we show the results for random MAX-3-SAT with 50 and 70 variables. In MAX-3-SAT, `Basic` and `Star` take the same profit of failed literal detection.

Finally, in Figure 4.10, we show the results for MAX-CUT. In this case, a similar behavior as for random MAX-2-SAT and MAX-3-SAT is observed, with `Basic` taking more advantage of failed literal detection than `Star`.

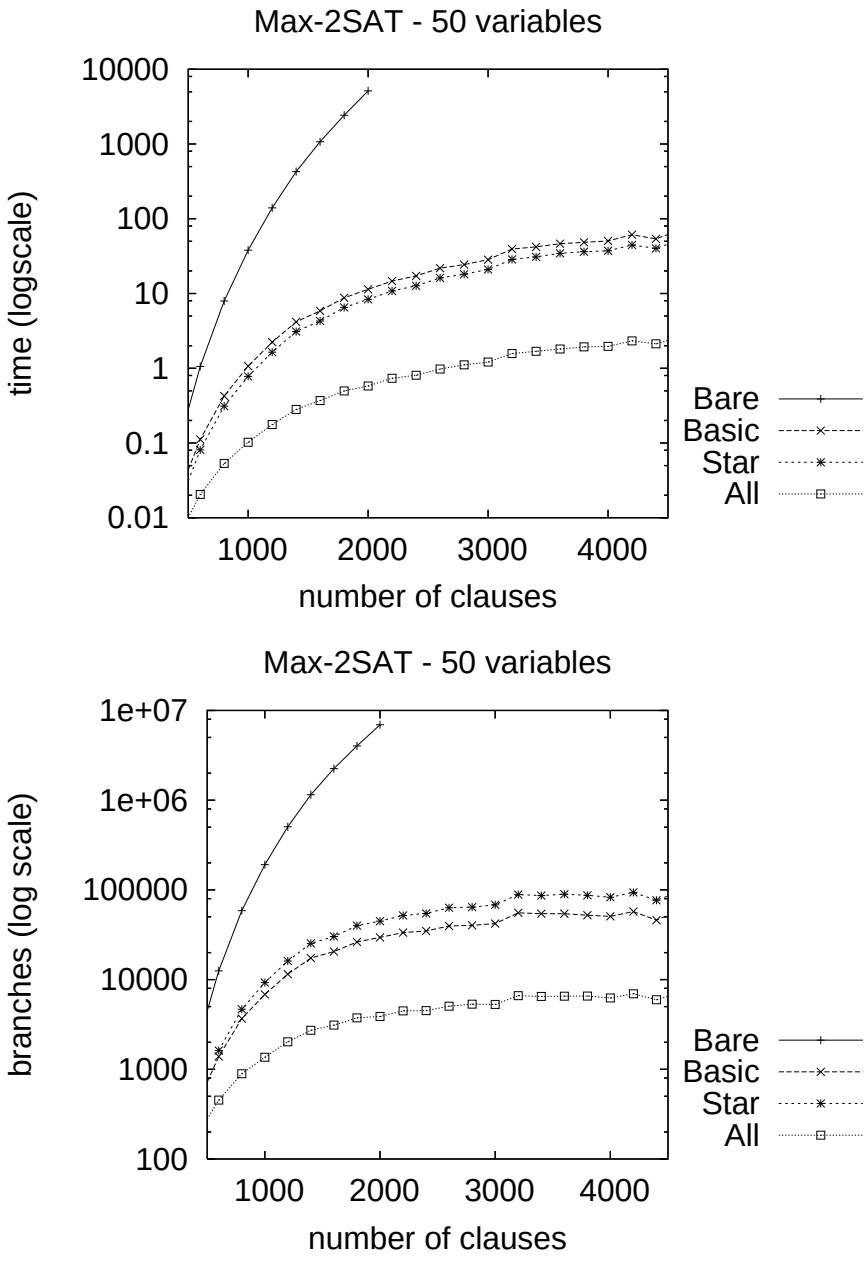


Figure 4.1: Random MAX-2-SAT with 50 variables

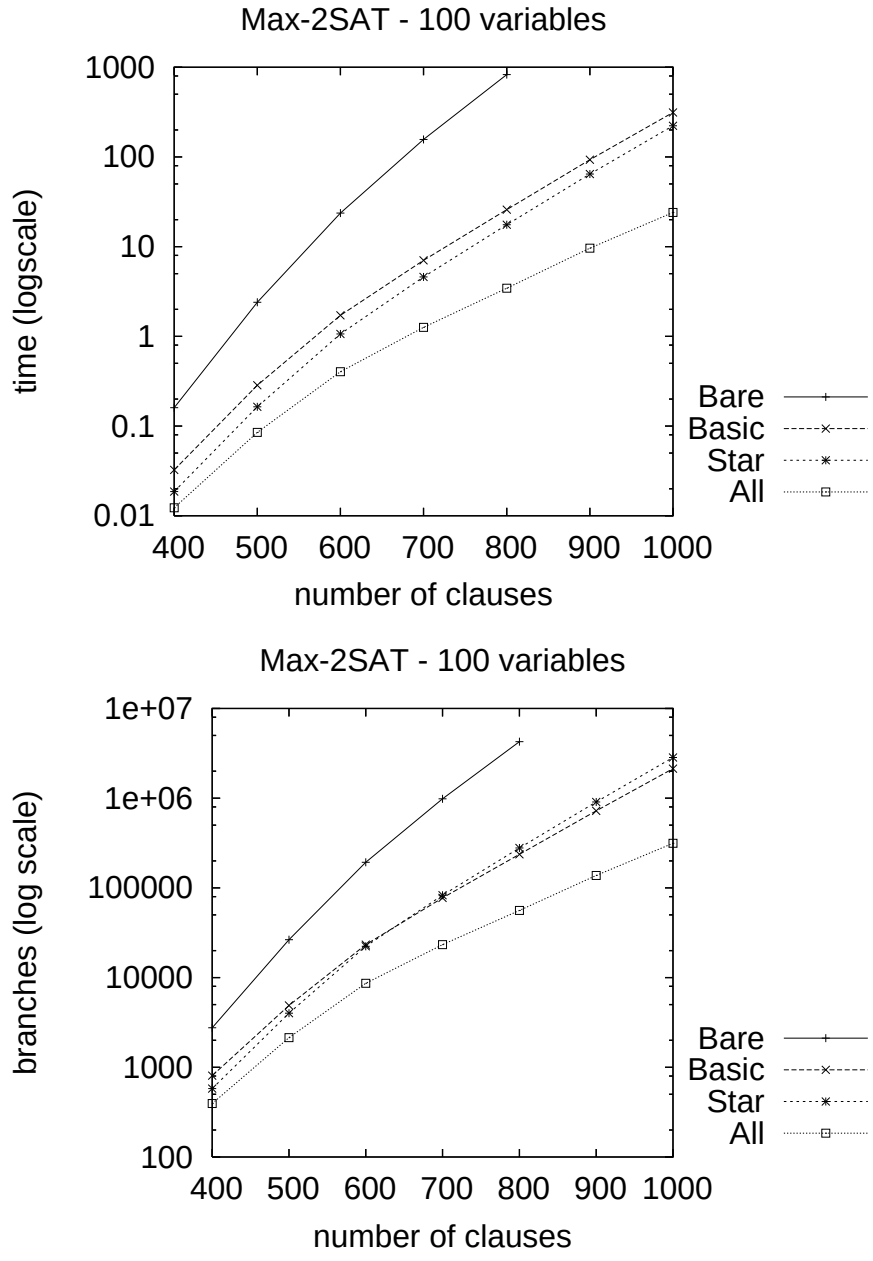


Figure 4.2: Random MAX-2-SAT with 100 variables

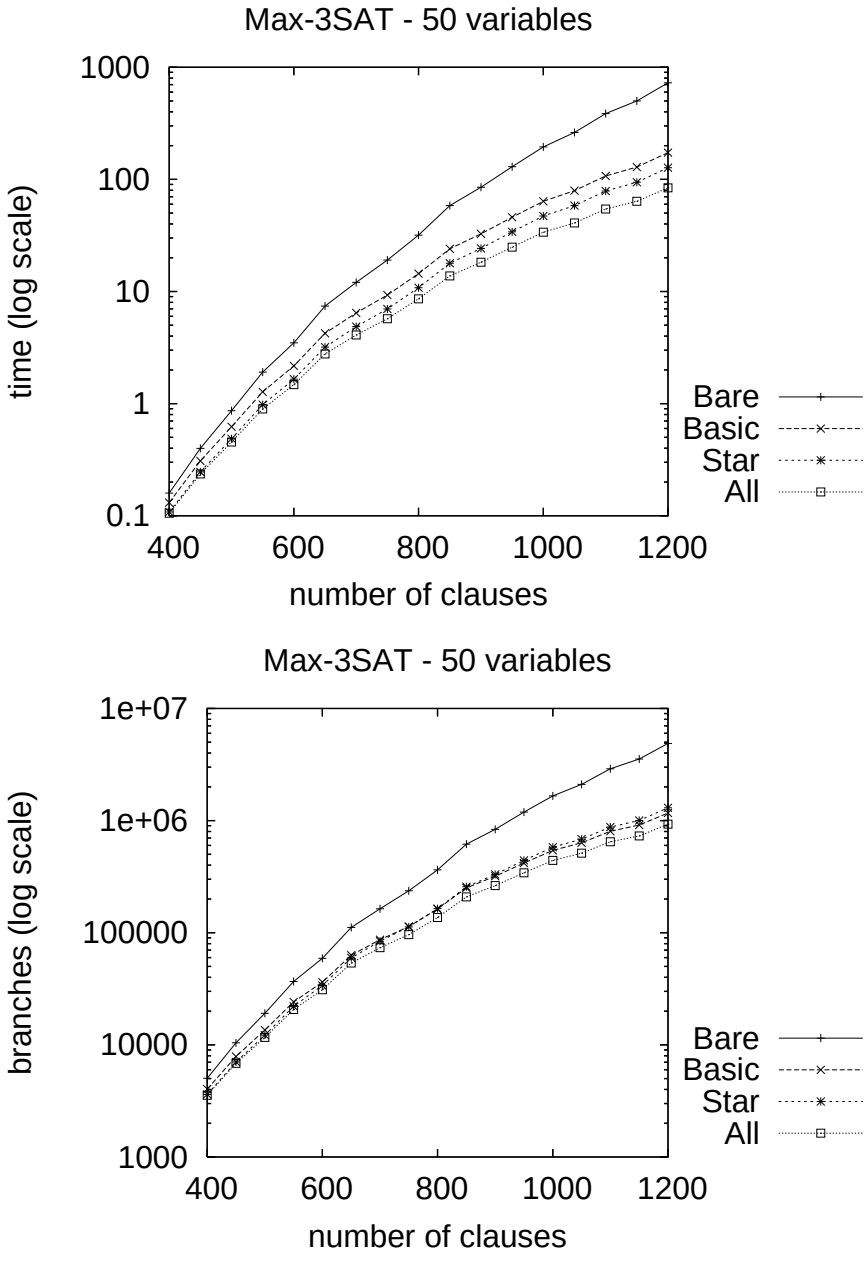


Figure 4.3: Random MAX-3-SAT with 50 variables

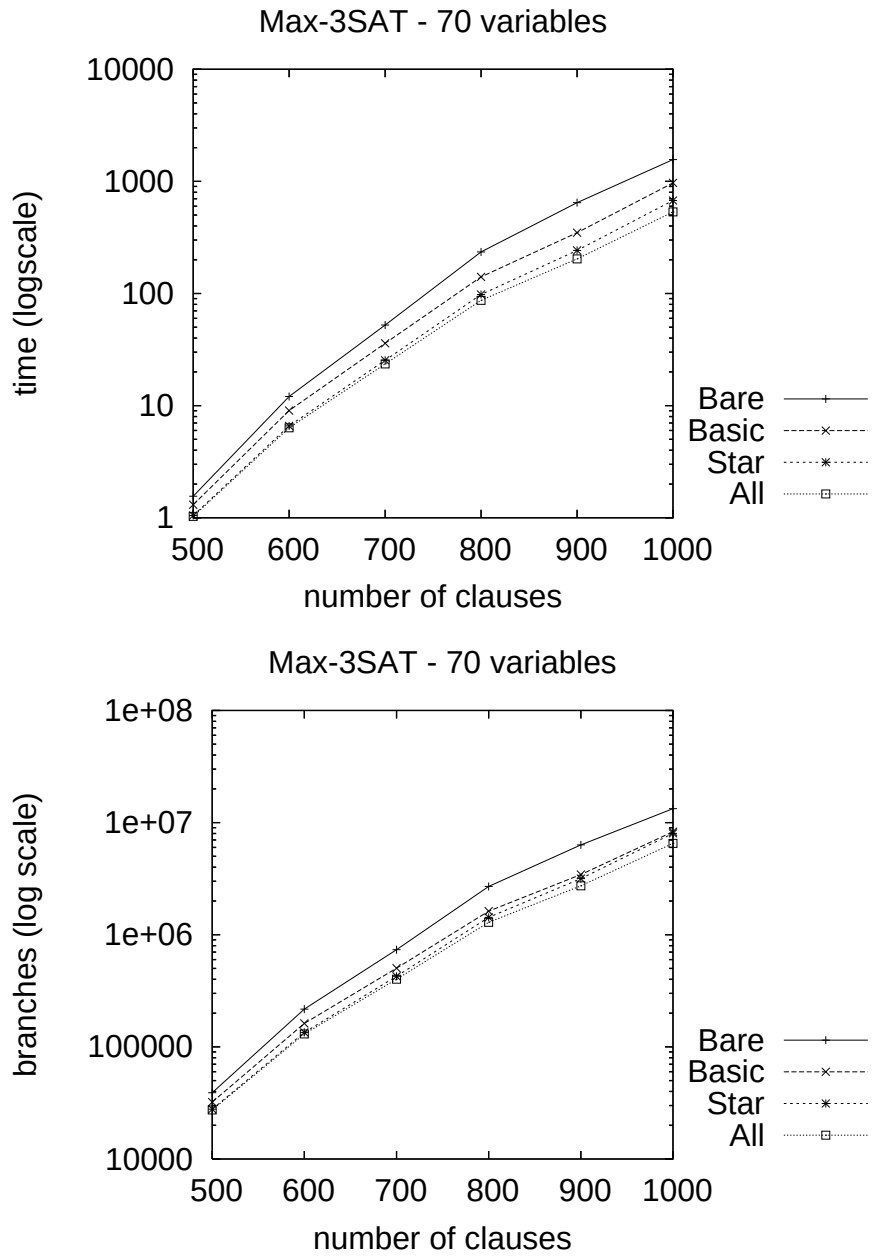


Figure 4.4: Random MAX-3-SAT with 70 variables

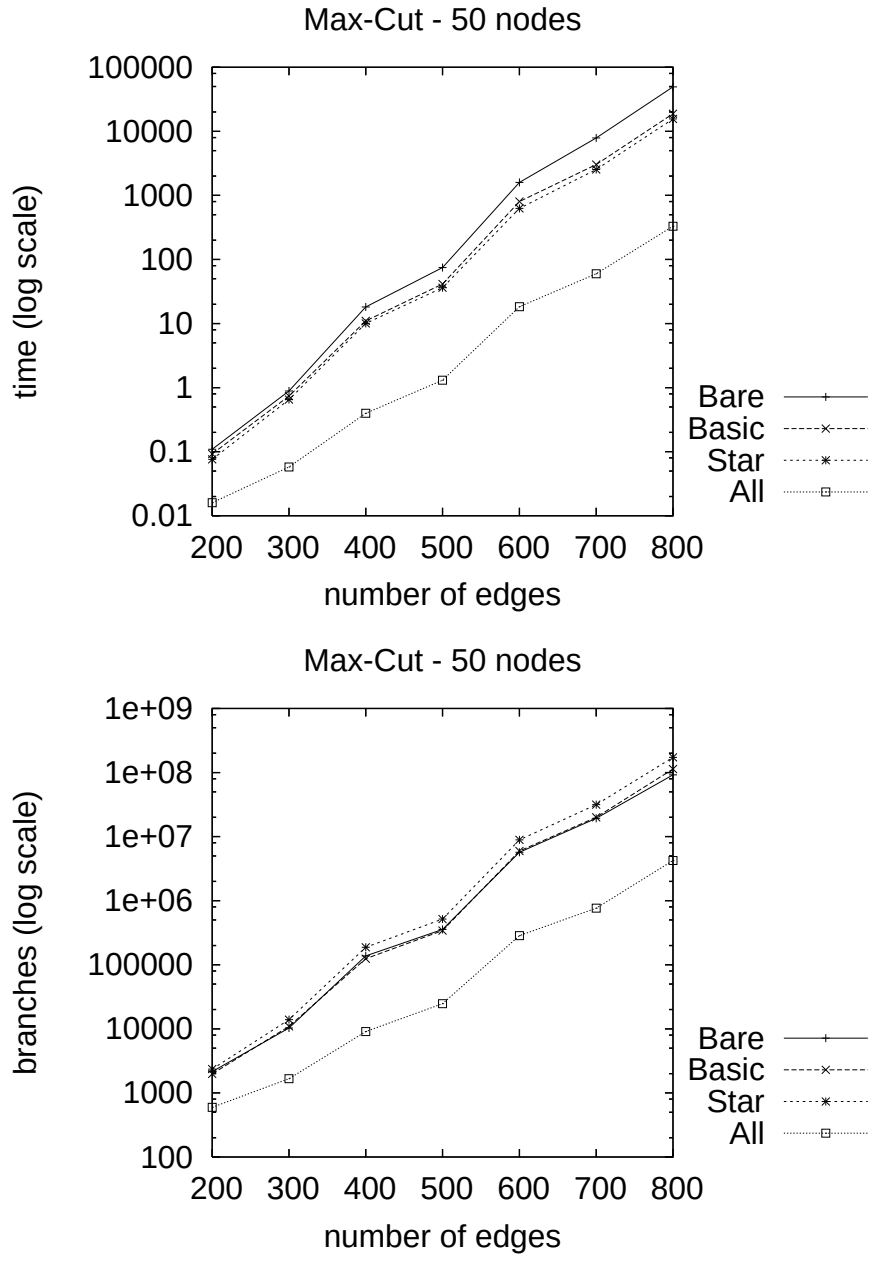


Figure 4.5: Random MAX-CUT with 50 variables

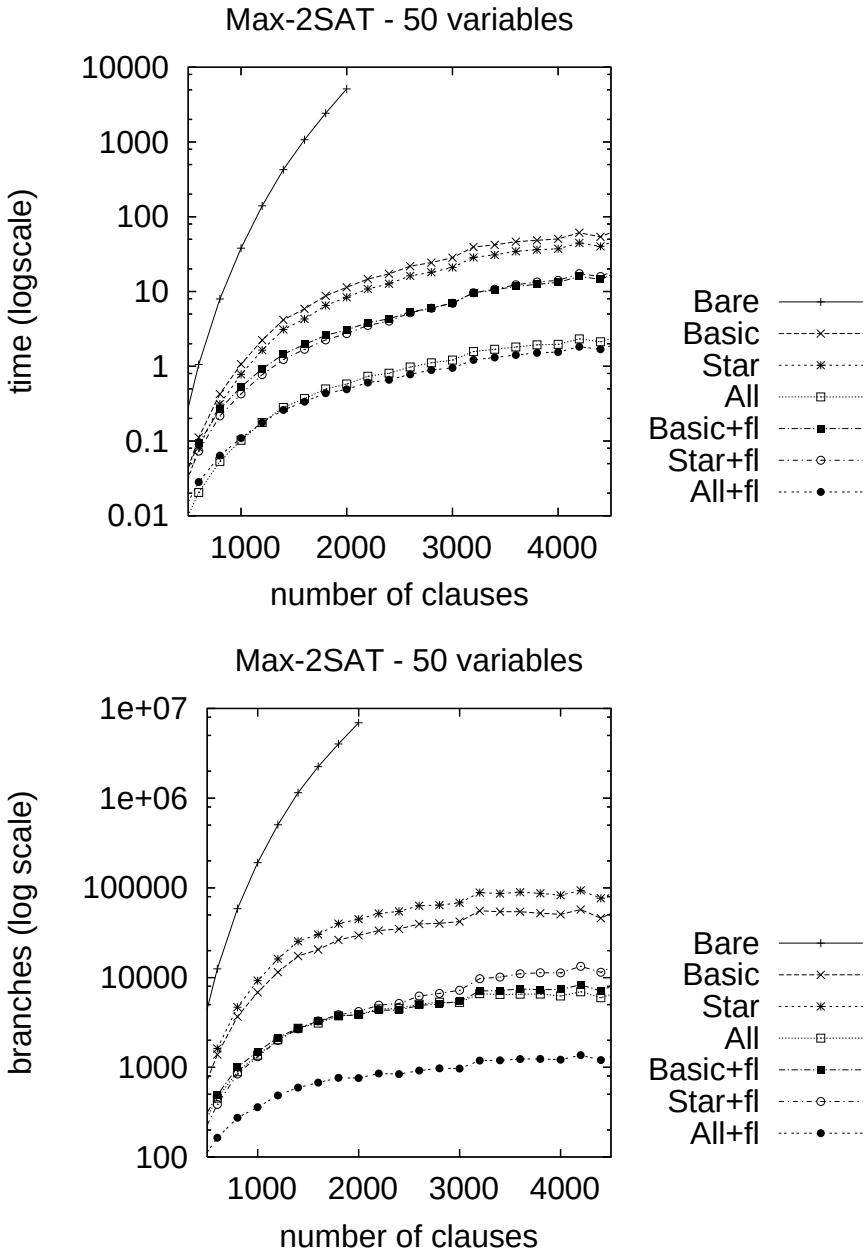


Figure 4.6: Random MAX-2-SAT 50 variables

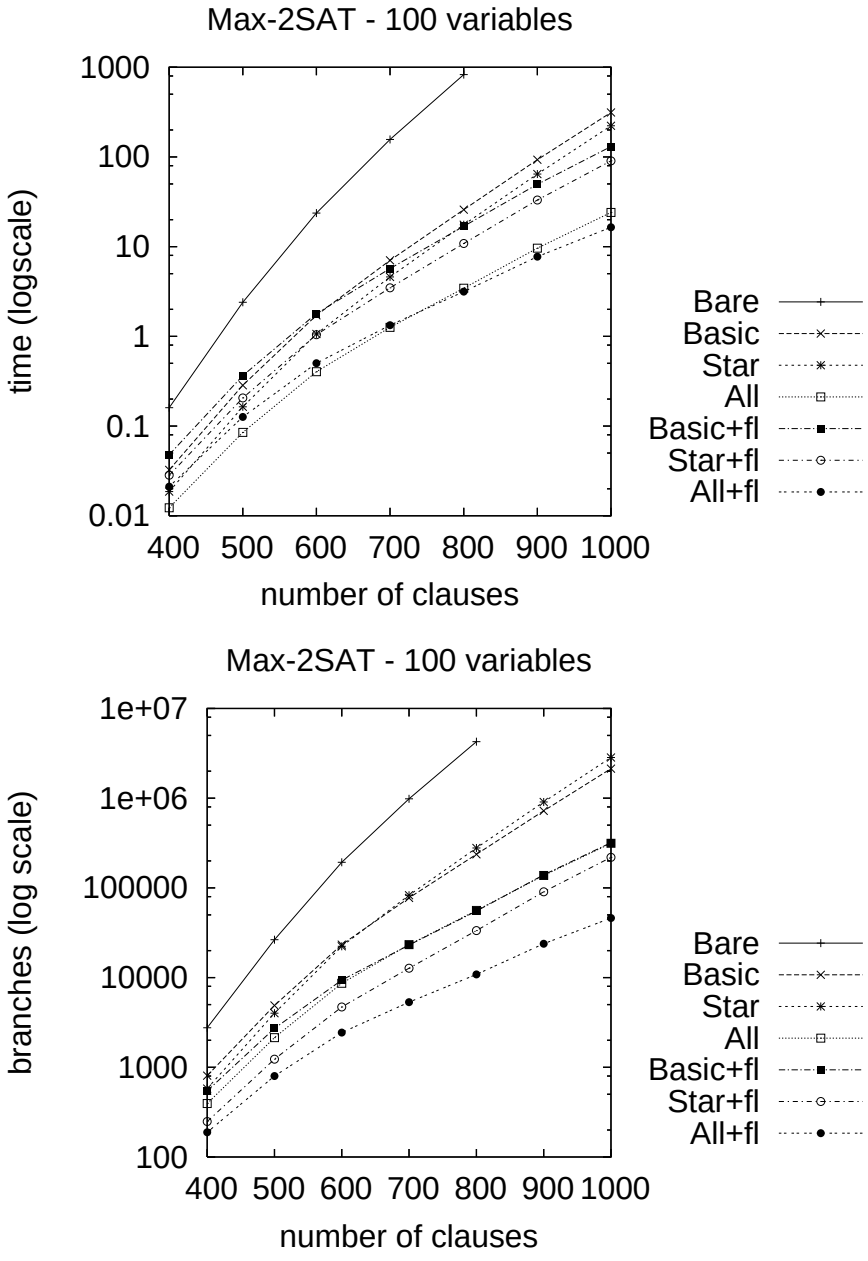


Figure 4.7: Random MAX-2-SAT 100 variables

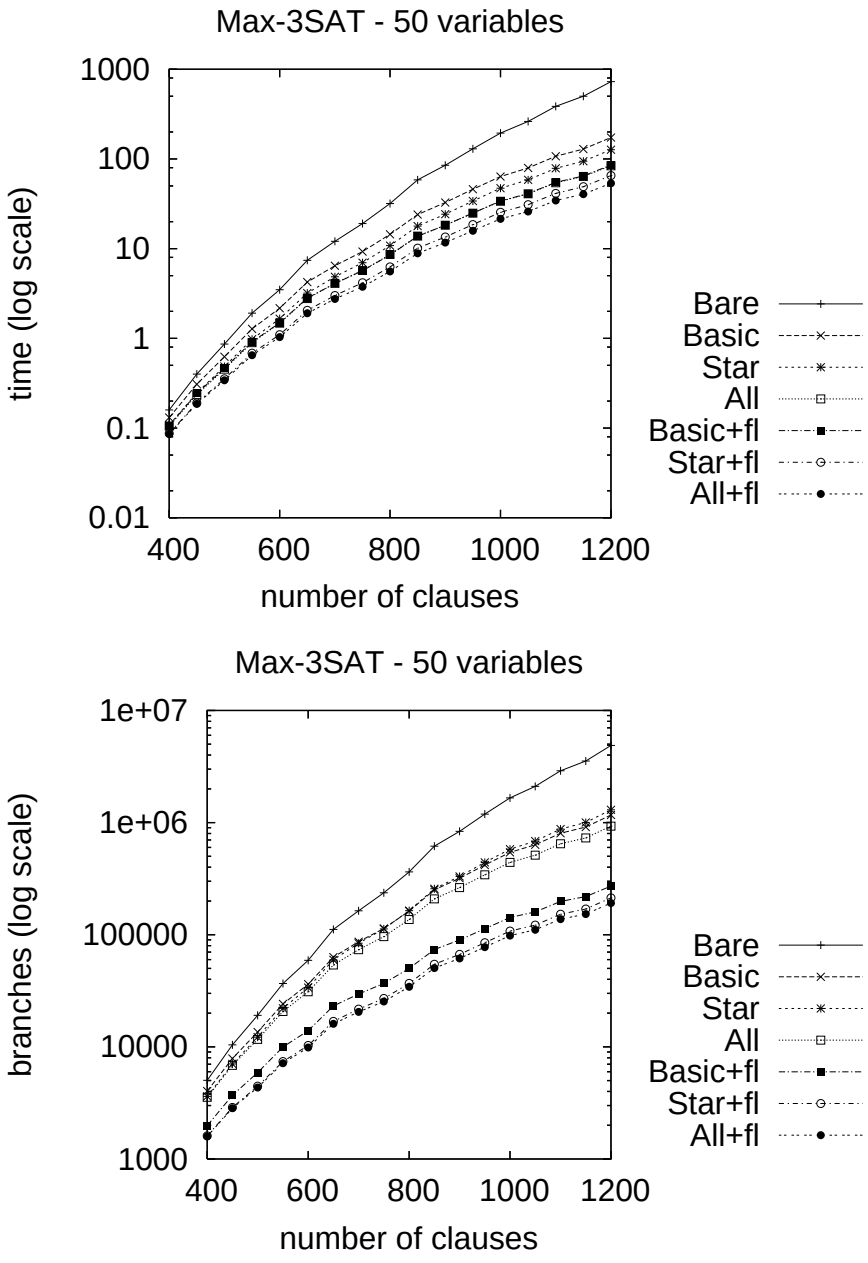


Figure 4.8: Random MAX-3-SAT 50 variables

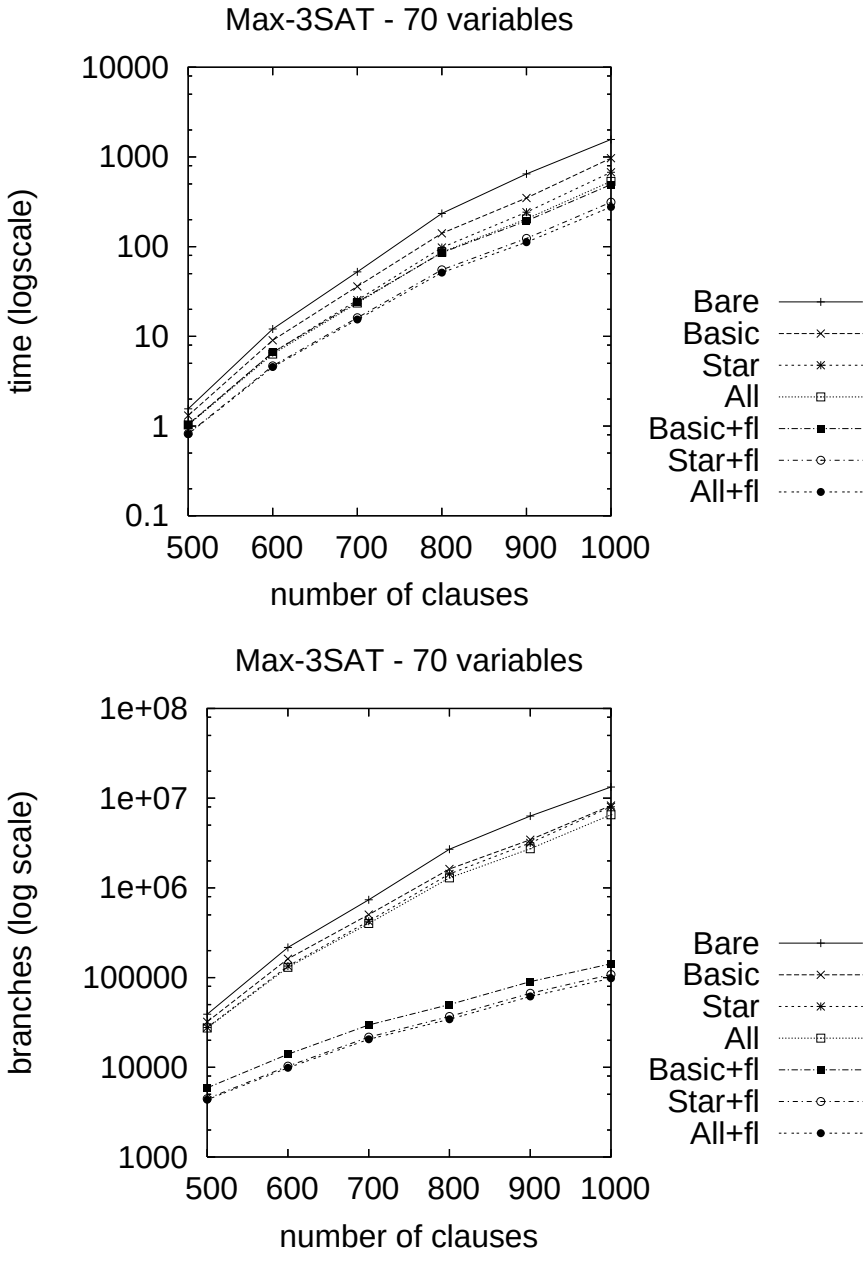


Figure 4.9: Random MAX-3-SAT 70 variables

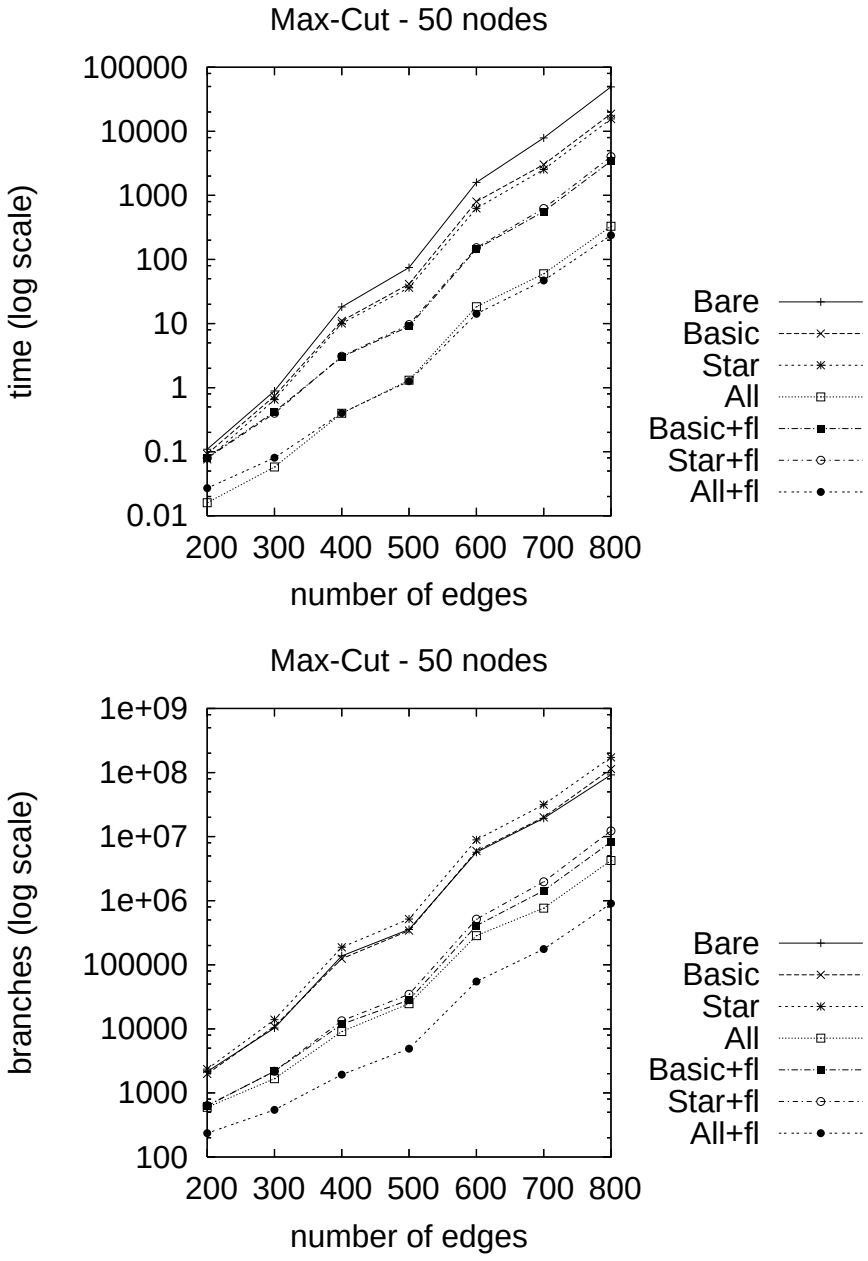


Figure 4.10: MAX-CUT

The behavior of Rule 4.3 and Rule 4.4 can be explained by two facts: (i) every application of Rule 4.3 and Rule 4.4 consumes two unit clauses but only gives one empty clause, limiting unit propagation in detecting more conflicts in subsequent search; and (ii) Rule 4.3 and Rule 4.4 add clauses which may contribute to detect further conflicts.

Looking carefully at Figure 4.2 and Figure 4.3, we observe that this behavior does not always occur. Depending on the number of clauses (or more precisely, the clause to variable ratio) in a formula, these two factors have different importance. When there are few clauses, unit propagation does not easily derive a contradiction from a unit clause, and the binary clauses added by Rule 4.3 and Rule 4.4 are important for deriving additional conflicts and improving the lower bound. This makes the search tree of `Star` smaller than the search tree of `Basic` (e.g. MAX-2-SAT instances of 100 variables and fewer than 600 clauses, MAX-3-SAT instances of 50 variables and fewer than 800 clauses). On the contrary, when there are many clauses, unit propagation easily derives a contradiction from a unit clause, so that the two unit clauses consumed by Rule 4.3 and Rule 4.4 probably would allow to derive two disjoint inconsistent subsets of clauses. In addition, the binary clauses added by Rule 4.3 and Rule 4.4 are relatively less important for deriving additional conflicts, considering the large number of clauses in the formula. In this case, the search tree of `Star` is larger than the search tree of `Basic`. We have seen that the rules have different behavior looking to the number of branches. By contrast, looking to the running time `Star` is always faster than `Basic`, which means that the incremental lower bound computation due to Rule 4.3 and Rule 4.4 is very effective, since the re-detection of many conflicts is avoided thanks to Rule 4.3 and Rule 4.4.

Rule 4.5 and Rule 4.6 do not limit the lower bound based on unit propagation in detecting more conflicts, since their application produces one empty clause and consumes just one unit clause, which allows to derive at most one conflict in any case. The added ternary clauses allow to improve the lower bound, so that the search tree of `All` is substantially smaller than the search tree of `Star`. The incremental lower bound computation due to Rule 4.5 and Rule 4.6 also contributes to the time performance of `All`. For example, while the search tree of `All` for instances with 50 variables and 2000 clauses is about 11.5 times smaller than the search tree of `Star`, `All` is 14 times faster than `Star`.

Although the rules do not involve ternary clauses, they are also powerful for MAX-3-SAT. Similarly to MAX-2-SAT, Rule 4.3 and Rule 4.4 slightly improve the lower bound when there are relatively few clauses, but do not improve the lower bound when the number of clauses increases. They improve the time performance thanks to the incremental lower bound computation they allowed. The gain increases as the number of clauses increases. For example, for problems with 70 variables, when the number of clauses is 600, `Star` is 36% faster than `Basic` and, when the number of clauses is 1000, the gain is 44%. Rule 4.5 and Rule 4.6 improve both the lower bound and the time performance of `All`. The gain increases as the number of clauses increases.

The intuitive explanation that the rules do not limit the lower bound can be credited observing the behavior of the solvers with failed literal detection. In Figure 4.6, `Star` consumes more branches than `Basic` while both spend approximately the same time. When increasing the number of clauses, the effect dramatically increases, `Basic` takes more advantage of the improved lower bound than `Star`.

As MAX-CUT is a MAX-2-SAT problem, we observe that the rules allow us to solve MAX-CUT instances much faster. Rule 4.3 and Rule 4.4 do not improve the lower bound when there are many clauses, but improve the time performance due to the incremental lower bound computation they allowed. Rule 4.5 and Rule 4.6 are more powerful than Rule 4.3 and Rule 4.4 for these instances, which only contain binary clauses but have some structure. In addition, the reduction of the tree size due to Rule 4.5 and Rule 4.6 contributes to the time performance of `All` more than the incrementality of the lower bound computation. For example, the search tree of `All` for instances with 800 edges is 40 times smaller than the search tree of `Star`, and `All` is 47 times faster.

In the next experiment, we compared different inference rules on the benchmarks submitted to the Max-SAT Evaluation 2006. Solvers ran in the same conditions as in the evaluation. In Table 4.1, the first column is the name of the benchmark set, the second column is the number of instances in the set, and the rest of columns display the average time, in seconds, needed by each solver to solve an instance (the number of solved instances in brackets). The maximum time allowed to solve an instance was 30 minutes.

In these experiments, it is clear that `Basic` is better than `Bare`, `Star` is better than `Basic`, and `All` is better than `Star`. For example, `All` solves three MAX-CUT Johnson instances within the time limit, while other solvers solve only two. The average time for `All` to solve one of these three instances is 44.46 seconds, the third instance needing more time to be solved than the first two ones.

4.5 Summary

We have introduced several inference rules that improve the performance of a MAX-SAT branch and bound solver. The rules transform a CNF formula into an equivalent formula with a larger number of empty clauses.

There are three important points worth to mention:

1. The inference rules do not need to make use of dynamic memory, because the transformed formulas do not have a greater number of literals.

2. The implication graph created by the lower bound UP (cf. Chapter 3) is used to apply the inference rules.

3. The transformations done to the formula by the inference rules are preserved until the algorithm backtracks to the current node, in contrast with the lower bound computation that requires to undo the transformations before any future variable assignment.

Thanks to them, these rules bring the creation of powerful MAX-SAT solvers, as will be shown in Chapter 6.

Set Name	#Instances	Bare	Basic	Star	All
MAX-CUT brock	11	401.47(9)	265.07(11)	215.40(11)	**13.17(11)**
MAX-CUT c-fat	7	1.92 (5)	3.11 (5)	2.84 (5)	**0.07(5)**
MAX-CUT hamming	6	39.42(2)	29.43(2)	29.48(2)	**171.30(3)**
MAX-CUT johnson	4	14.91(2)	8.57 (2)	7.21 (2)	**44.46(3)**
MAX-CUT keller	2	512.66(2)	213.64(2)	163.26(2)	**6.82(2)**
MAX-CUT p hat	12	72.16(9)	286.09(12)	226.24(12)	**16.81(12)**
MAX-CUT san	11	801.95(7)	305.75(7)	245.70(7)	**258.65(11)**
MAX-CUT sanr	4	323.67(3)	134.74(3)	107.76(3)	**71.00(4)**
MAX-CUT max cut	40	610.28(35)	481.48(40)	450.05(40)	**7.18(40)**
MAX-CUT SPINGLASS	5	0.22 (2)	0.19 (2)	0.15 (2)	**0.14(2)**
MAX-ONE	45	**0.03 (45)**	**0.03 (45)**	**0.03 (45)**	**0.03(45)**
RAMSEY	48	8.93 (34)	8.42 (34)	7.80 (34)	**7.78(34)**
MAX2SAT 100VARS	50	95.01(50)	11.30(50)	8.14 (50)	**1.25(50)**
MAX2SAT 140VARS	50	153.28(49)	51.76(50)	34.14(50)	**6.94(50)**
MAX2SAT 60VARS	50	1.35 (50)	0.08 (50)	0.06 (50)	**0.02(50)**
MAX2SAT DISCARDED	180	126.98(162)	71.85(173)	68.97(175)	**22.72(180)**
MAX3SAT 40VARS	50	11.52(50)	3.33 (50)	2.52 (50)	**1.92(50)**
MAX3SAT 60VARS	50	167.17(50)	72.72(50)	52.14(50)	**40.27(50)**

Table 4.1: Rule evaluation by benchmarks in the MAX-SAT Evaluation 2006.

Set Name	#Instances	Bare	Basic+fl	Star+fl	All+fl
MAX-CUT brock	11	401.47(9)	85.07(11)	87.73(11)	**12.50(11)**
MAX-CUT c-fat	7	1.92 (5)	0.35 (5)	0.29 (5)	**0.07 (5)**
MAX-CUT hamming	6	39.42(2)	4.03 (2)	4.57 (2)	**179.65(3)**
MAX-CUT johnson	4	14.91(2)	577.53(3)	583.82(3)	**45.44(3)**
MAX-CUT keller	2	512.66(2)	62.86(2)	67.21(2)	**6.06 (2)**
MAX-CUT p hat	12	72.16(9)	99.12(12)	96.45(12)	**15.73(12)**
MAX-CUT san	11	801.95(7)	94.67(7)	97.20(7)	**273.65(11)**
MAX-CUT sanr	4	323.67(3)	413.95(4)	402.46(4)	**71.70(4)**
MAX-CUT max cut	40	610.28(35)	78.47(40)	88.88(40)	**5.54 (40)**
MAX-CUT SPINGLASS	5	0.22 (2)	44.73(3)	**44.53(3)**	**44.53(3)**
MAX-ONE	45	0.03 (45)	0.06 (45)	**0.02 (45)**	**0.02 (45)**
RAMSEY	48	8.93 (34)	11.34(34)	8.96 (34)	**8.92 (34)**
MAX2SAT 100VARS	50	95.01(50)	9.22 (50)	6.39 (50)	**1.39 (50)**
MAX2SAT 140VARS	50	153.28(49)	45.73(50)	24.36(50)	**6.91 (50)**
MAX2SAT 60VARS	50	1.35 (50)	0.10 (50)	0.07 (50)	**0.03 (50)**
MAX2SAT DISCARDED	180	126.98(162)	61.74(176)	60.89(179)	**16.38(180)**
MAX3SAT 40VARS	50	11.52(50)	2.37 (50)	1.72 (50)	**1.50 (50)**
MAX3SAT 60VARS	50	167.17(50)	46.58(50)	26.74(50)	**23.35(50)**

Table 4.2: Rule evaluation by benchmarks in the MAX-SAT Evaluation 2006 with failed literal detection

Chapter 5

Implementing a weighted MAX-SAT solver

The MAX-SAT formalism can be extended to weighted MAX-SAT to facilitate the solving of optimization problems having constraints with different importance (e.g., Max-Clique converted to a CNF formula). Weighted MAX-SAT has a richer expressivity which can be exploited in the search.

Let us see an example to show the difference: Let ϕ be a CNF formula with four clauses $p \vee \neg q$, $p \vee \neg r$, $q \vee r$, $\neg p$. As a MAX-SAT instance, an optimal solution is the assignment $p = false$, $q = false$, $r = true$, with 1 unsatisfied clause. If we want an assignment satisfying the two first clauses and the maximum number of the rest clauses, in MAX-SAT we might add two copies of the first two clauses. When the problem becomes larger, the number of repeated clauses increases. To avoid such a situation, a weight can be associated to each clause having the weighted CNF formula: $(p \vee \neg q, 3)$, $(p \vee \neg r, 3)$, $(q \vee r, 1)$, $(\neg p, 1)$. A weighted MAX-SAT solution is the assignment $p = true$, $q = false$, $r = false$.

Lower bounds and inference rules for MAX-SAT can be naturally extended to Weighted MAX-SAT. We have performed many of such extensions, and implemented them in solver `Lazy` [AMP04a], which was created at the beginning of this research. Its name comes from an original lazy data structure with a static variable selection heuristic that speeds up the search at the price of delaying the evaluation of the clauses. Although `Lazy` lacks the most powerful techniques for solving weighted MAX-SAT, it is interesting to see the behavior of its weighted MAX-SAT rules, and the performance of a static variable selection heuristic.

In the rest of the chapter we define a lower bound and a set of inference rules for weighted MAX-SAT, describe their implementation in the solver `Lazy`, and report on an experimental evaluation that provides empirical evidence of the performance of the rules.

5.1 Basic equivalences for weighted MAX-SAT

In the forthcoming sections, we will use three equivalence properties:

Property 5.1 *Clauses with null weight can be removed from the formula.*

Property 5.2 *Two weighted clauses* $(c,\ w_1)$ *and* $(c,\ w_2)$ *are equivalent to the weighted clause* $(c,\ w_1 + w_2)$.

Property 5.3 *Let* ϕ *be a weighted CNF formula, let* w *be the minimum weight in all the clauses of* ϕ*, let* ϕ_w *be* ϕ *with all its clauses with weight* w*, and let* $\overline{\phi_w}$ *be* ϕ *where every clause* $(c_i, w_i) \in \phi$ *becomes* $(c_i, w_i - w)$ *(i.e.,* $\phi = \phi_w \cup \overline{\phi_w}$*). Then, an unweighted rule can be applied* w *times to* ϕ_w*, and* $\overline{\phi_w}$ *remains unchanged.*

Lower bounds and inference rules applied to MAX-SAT can be extended to weighted MAX-SAT using Property 5.3.

Example 5.1 *Let* ϕ *be the weighted CNF formula* $(l_1 \vee l_2, 5),\ (\bar{l}_1, 4),\ (\bar{l}_2, 3)$*. Having the above properties and applying Rule 4.3, we obtain the following weighted CNF formula* $(\square, 3),\ (\bar{l}_1 \vee \bar{l}_2, 3),\ (l_1 \vee l_2, 2),\ (\bar{l}_1, 1)$.

5.2 Lazy solver

We introduce a DLL-based branch and bound algorithm to solve weighted MAX-SAT, called `Lazy`. In previous chapters, we have seen that a MAX-SAT algorithm keeps track of the number of unsatisfied clauses. By contrast, a weighted MAX-SAT algorithm keeps track of the sum of the weights of unsatisfied clauses.

`Lazy` has a static variable selection heuristic and lazy data structures. The variable selection heuristic actually defines an order of the variables to be assigned before the branch and bound algorithm is executed. The algorithm scheme is drawn in Algorithm 5.1.

Algorithm 5.1: `SolverLazy`(ϕ, i) : Branch and Bound in `Lazy`

Output: Minimum sum of weights of unsatisfied clauses by any assignment of ϕ

Function *`SolverLazy`(ϕ : CNF formula, i : Natural)* **: *Natural***

```
▷ p_i IS THE iTH VARIABLE TO BE ASSIGNED ◁
if φ = ∅ or φ only contains empty clauses then
    return EmptyClauses(φ)
φ ← InferenceRules(φ)
if LowerBound(φ) ≥ UpperBound(φ) then
    return UpperBound(φ)
return min( SolverLazy( φ_{p_i}, i + 1), SolverLazy( φ_{¬p_i}, i + 1) )
```

The solver computes a lower bound and applies several inference rules. The lower bound is a weighted version of lower bound UP for MAX-SAT (cf. Section 3.3), and takes into account the static variable selection heuristic. In order to make this estimation efficient for `Lazy`, the computation is restricted to binary clauses. The lower bound computed in `Lazy` [ZSM03b] is:

1. For every weighted unit clause (p, w_1) and for every binary clause of the form $(\neg p \vee q, w_2)$ in ϕ, add a unit clause (q, w) into ϕ, and substract a weight w to unit clause (p, w_1), where $w = \min(w_1, w_2)$.

2. If a conflict is found, increase the lower bound by w.

3. If there are not more unit clauses to be propagated, restore the original ϕ and return the lower bound.

`Lazy` implements also two efficient inference rules, the weighted versions of rules CUC and ACC:

Complementary Unit Clause (CUC) rule Let $\phi_1 = (\ell, w_1), (\bar{\ell}, w_2) \cup \phi'$ be a weighted CNF formula. Then, ϕ_1 is equivalent to the weighted CNF formula $\phi_2 = (\square, w), (\ell, w_1 - w), (\bar{\ell}, w_2 - w) \cup \phi'$, where $w = \min(w_1, w_2)$. Clauses with null weight are removed. Then, ϕ_1 and ϕ_2 are equivalent.

The soundness of the rule follows from the soundness of the unweighted rule CUC (i.e., Rule 4.2), and the application of Property 5.3.

Almost Common Clause (ACC) rule as Preprocessing Let $\phi_1 = (l_1 \vee l_2, w_1), (\bar{l}_1 \vee l_2, w_2) \cup \phi'$ be a weighted CNF formula, $w = \min(w_1, w_2)$, and $\phi_2 = (l_2, w), (l_1 \vee l_2, w_1 - w), (\bar{l}_1 \vee l_2, w_2 - w) \cup \phi'$ be a weighted CNF formula. Clauses with weight null are removed. Then, ϕ_1 and ϕ_2 are equivalent.

The soundness of the rule follows from the soundness of the unweighted rule ACC (i.e., Rule 4.1), and the application of Property 5.3.

This rule was applied as a preprocessing due to its expensive computational cost in `Lazy`.

5.2.1 Data structures

Similar to the counter based data structure in SAT solvers (cf. Section 2.2.3), most MAX-SAT branch and bound algorithms represent clauses as lists of literals with counters, and associate with each variable p a list of the clauses that contain literal p or literal $\neg p$. Clearly, after assigning variable p, the clauses with those literals are immediately aware of the assignment of p. In general, we use the term adjacency lists to refer to data structures in which each variable p contains a complete list of the clauses that contain a literal p or a literal $\neg p$.

As was reported in [Lyn04], adjacency list based data structures share a common problem: each variable p keeps references to a potentially large number of clauses. Clearly, this impacts negatively the amount of operations associated

with assigning p. Moreover, it is often the case that most of the clause references of p do not need to be analyzed when p is assigned, since most of the clauses do not become unit or unsatisfied. Observe that *lazily* declaring a clause to be satisfied does not affect the correctness of the algorithm.

In solver `Lazy`, we define data structures with three levels, each one containing clauses of increasing size:

Unit clauses are stored in a vector of integers, where each position represents the weight of a literal.

Binary clauses are stored in lists of lists. For every literal l_i there is a list of pairs $(literal, weight)$. A pair (l_j, w) in the list of literal l_i represents a binary clause $(l_i \vee l_j, w)$. The variable in literal l_i is assigned before the variable in literal l_j.

Larger clauses are stored in a list of clauses, ordered by their antepenultimate literal.

Every time a literal ℓ takes value true, the following operations are performed:

1. The weight of unit clause $\{\bar{\ell}\}$ is added to the empty clause. This addition is saved for backtracking.
2. For every pair (l_2, w) in the list of binary clauses of literal $\bar{\ell}$, the weight w is added to unit clause l_2. This addition is saved for backtracking.
3. For every larger clause $(l_1 \vee l_2 \vee \bar{\ell} \vee \cdots, w)$ with antepenultimate literal ℓ, the clause is evaluated. If the clause is not satisfied, the clause $(l_1 \vee l_2, w)$ is added to the set of binary clauses. The new clause is added at the end of the list to make the backtracking faster (amortized linear time over the number of clauses).

The laziness of this data structure is in the set of *larger clauses*, where clauses are ordered by their antepenultimate variable following the order used to instantiate variables, and three references: one to the antepenultimate literal, one to the penultimate literal, and one to the last literal of the clause. When a variable p is fixed to true (false), the clauses whose antepenultimate literal is $\neg p$ (p) are evaluated. If there is an instantiated literal in the clause which is satisfied, the clause becomes satisfied; otherwise, a binary clause with the same weight, whose literals are the penultimate and the last literal of the clause, is derived. Thus, given a clause with four literals, it is not necessary to perform any operation in that clause until two of the literals have been instantiated; i.e., the evaluation of a clause with k literals can be delayed until $k-2$ literals have been instantiated.

5.2.2 Variable selection heuristic

The variable selection heuristic in `Lazy` is static, and computed before the branch and bound algorithm. The heuristic performs the ordering in two phases: In the first phase, it orders the literals by the sum of weights associated to their

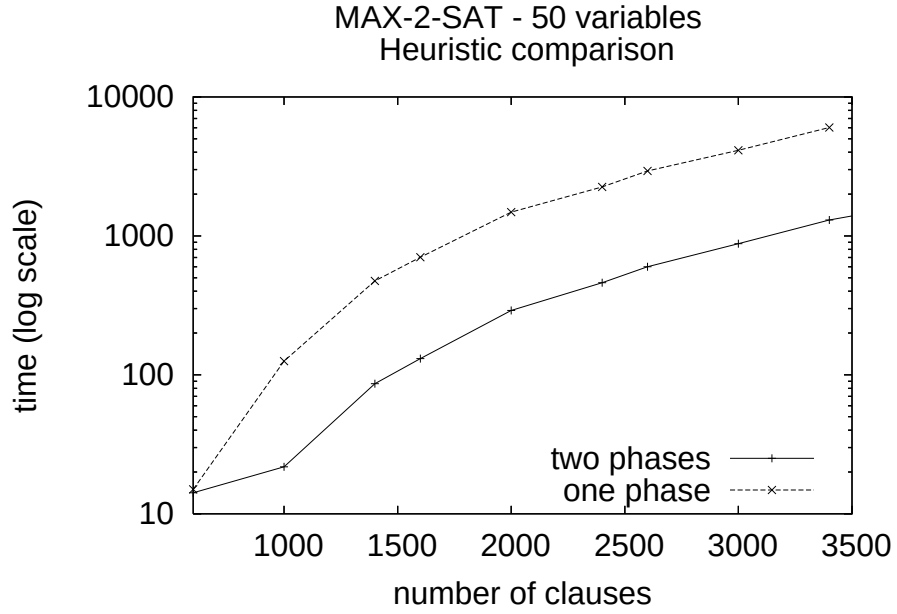


Figure 5.1: Comparison of applying the first phase only and the two phases in the variable selection heuristic.

clauses. In the second phase, the weight of a variable is incremented in a directly proportional manner to its neighboring literals weights. A literal l_1 is a neighbor of literal l_2 if there exists a clause having literals l_1 and $\bar{l}_2$. A variable p is a neighbor of literal ℓ, if literals p or $\neg p$ are [NLBH+04]. Such an ordering method is similar to SAT variable selection heuristic Backbone [DD01] constrained to only one level.[1] This second phase is designed for breaking ties in the first phase, which often occur in random MAX-k-SAT. The difference of application of one phase and two phases is shown in Figure 5.1 for random weighted MAX-2-SAT.

We illustrate the variable selection heuristic with the example below. For clearness, we used an unweighted formula.

Example 5.2 *Let variable p have 4 occurrences and two neighboring variables p_1 and p_2 with 2 occurrences each variable. Let variable q have also 4 occurrences and two neighboring variables q_1 and q_2 with 3 occurrences each variable. The instantiation of any of the two variables brings 4 unit clauses, nevertheless variable q brings clauses with variables with more occurrences. After instantiating variable q, variables q_1 and q_2 are probably going to be instantiated. The assignment of q makes the problem have more short clauses, therefore making it easier to be solved [SW02].*

[1] A method of selecting variables in MAX-SAT using the backbone is found in [ZRL03].

5.3 Empirical evaluation

We first define the benchmarks used in the experimental evaluation: random formulas, graph coloring, and MAX-ONES. The last two problems have been used in order to check the solver with instances having structure. Then, we report on the experimental results over the previous problems, and all the weighted benchmarks of the MaxSAT Evaluation 2006.

5.3.1 Benchmarks

In the experimentation, we solve random weighted MAX-k-SAT problems; and two problems that can be reduced to weighted MAX-SAT: Graph coloring and MAX-ONES.

For the random weighted MAX-k-SAT instances generation, we have modified generator `mwff` (refer to Section 3.5.1) with the addition of a random integer weight for each clause. The integer is in the range $[1-10]$. For the random MAX-ONES instances generation, the same solver has been modified, following the transformation below.

For the sake of clearness, before mapping the two problems to weighted MAX-SAT, we will map each problem to an intermediate problem, *Partial MAX-SAT* [MIK96]. The CNF formula is split to two sets: the set of hard clauses, where every clause has to be satisfied, and the set of soft clauses, where the maximum number of clauses has to be satisfied. Then, partial MAX-SAT can be mapped to a *weighted MAX-SAT* instance. This is done applying the following rule:

Rule 5.1 *Given a partial MAX-SAT instance with hard clauses $c_{h_1}, \ldots, c_{h_m}$ and soft clauses $c_{s_1}, \ldots, c_{s_n}$, it can be transformed to a weighted MAX-SAT instance associating weight 1 to soft clauses and weight $n+1$ to hard clauses:*

$$c_{h_1}, \ldots, c_{h_m}, c_{s_1}, \ldots, c_{s_n} \Rightarrow (c_{h_1}, n+1), \ldots, (c_{h_m}, n+1), (c_{s_1}, 1), \ldots, (c_{s_n}, 1)$$

Graph coloring

Given an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, and a function $c : V \rightarrow K = \{1, 2, \ldots, k\}$, where K is the set of colors, such that $c(v_i) \neq c(v_j)$ for every edge $(v_i, v_j) \in E$.[2] The graph k-coloring problem is to determine the minimum number of edges to be removed that make the graph be colored with k colors.

In order to reduce a graph coloring instance to a partial MAX-SAT instance, each vertex v_i is reduced to $|K|$ variables, where $|K|$ is the number of colors we have. Thus, the CNF formula will have $|V| \cdot |K|$ Boolean variables, where $|V|$ is the number of vertices in G. Each Boolean variable will be denoted by x_{ik}, where i denotes a vertex and k a color. Variable x_{ik} will be assigned to true if the vertex i is assigned to color k. Otherwise, it is assigned to false.

[2] In other words, adjacent vertices must have different colors.

The constraints of the graph coloring problem will be mapped to clauses according to the three following sets:

- At Least One: to represent that a vertex receives at least a color is expressed by:
$$x_{i1} \vee \cdots \vee x_{ik} \qquad \forall v_i \in V$$
- At Most One: to represent that one vertex must have at most one assigned color is expressed by:
$$\neg x_{ik} \vee \neg x_{ik'} \quad \forall v_i \in V \qquad \forall k, k' \in K \quad (1 \leq k < k' \leq |K|)$$
- The coloring problem itself: to represent that two adjacent vertices must be displayed in different colors is expressed by:
$$\neg x_{ik} \vee \neg x_{i'k} \quad \forall (v_i, v_{i'}) \in E \qquad \forall k \in K \quad (1 \leq k \leq |K|)$$

In the partial MAX-SAT mapping, the first two sets are mapped as hard clauses, and the third one as soft clauses. Finally, the partial MAX-SAT instance is mapped to weighted MAX-SAT using Rule 5.1.

MAX-ONES

Another optimization problem derived from the SAT decision problem is MAX-ONES. The MAX-ONES problem is to find a satisfying truth assignment that maximizes the number of variables assigned with the value 1 (or true).

In order to apply the mapping to partial MAX-SAT, let ϕ be a CNF formula. All the clauses in ϕ are added to the set of hard clauses, and for each variable in ϕ a soft unit clause is added to the set of soft clauses with the variable in positive polarity. Finally, the partial MAX-SAT instance is mapped to weighted MAX-SAT using Rule 5.1.

Example 5.3 *Let $\phi = \{p \vee q\}, \{\neg p \vee q\}, \{p \vee \neg q\}$ be a CNF formula. The partial MAX-SAT instance solving the MAX-ONES problem for ϕ is formed by the set of hard clauses $\{p \vee q\}, \{\neg p \vee q\}, \{p \vee \neg q\}$, and the set of soft clauses $\{p\}, \{q\}$.*

5.3.2 Experimental results

In this section, in order to compare the lower bound and inference rules defined, we have used three simplified versions of `Lazy`:

Bare does not apply neither lower bound, nor inference rule.

LB applies the lower bound.

CUC applies inference rule CUC.

ACC applies inference rule ACC.

LB+CUC applies the lower bound and inference rule CUC.

LB+ACC applies the lower bound and inference rule ACC.

Lazy applies the lower bound and both inference rules.

In the first experiment, we observe the behavior of the seven solvers in random weighted MAX-2-SAT instances with 50 and 100 variables (see Figure 5.2 and Figure 5.3). We can see that the best solver is `Lazy`, and the worst is `Bare`. For 50 variables, the solvers can be divided in four groups, which share the same number of backtracks. Listing them from the worst to the best are:

- `Bare` and `CUC`;
- `ACC`, that has a slope different from the rest, improving its performance when the number of clauses increases;
- `LB` and `LB+CUC`; and
- `LB+ACC` and `Lazy`, that are the best ones.

We can see that the `CUC` rule does not change the number of backtracks, but improves the time spent, the `ACC` rule is more useful when the problem is more constrained, and the lower bound is useful in the middle of the plot. For 100 variables, the behavior is the same as at the beginning of the plot for 50 variables: the lower bound is very useful. It is expected that `ACC` will become more useful when the number of clauses is increased.

In the second experiment, we observe the performance of the five solvers for random weighted MAX-3-SAT with 50 and 70 variables (Figure 5.4 and Figure 5.5). Solvers with `ACC` have been discarded because the preprocessing is only applicable to MAX-2-SAT instances. We can see that the best solver is `Lazy` and the worst is `Bare`. In this case, the solvers collapse in two overlapped lines on the number of backtracks: in the first are `Bare` and `CUC`; and in the second `LB`, `LB+CUC` and `Lazy`. What makes the difference in the number of backtracks is the application of the lower bound. In running time, `CUC` helps to decrease it.

In the third experiment, we observe the performance of the five solvers for random graph coloring with density of 90% and increasing number of nodes (Figure 5.6). The influence of the lower bound and the inference rules is not so important as in weighted MAX-3-SAT. This can be confirmed by the fact that `CUC` is the fastest algorithm, because the lower bound does not help to reduce the time. In this case, the solvers also collapse in two overlapped lines on the number of backtracks, with the same solvers.

In the fourth experiment, we observe the performance of the seven solvers for random MAX-ONES with 50 and 70 variables (Figure 5.7 and Figure 5.8). The same behavior as in random weighted MAX-2-SAT can be observed for random MAX-ONES 2-SAT (hard clauses are binary), and the same as in random weighted MAX-3-SAT can be observed for random MAX-ONES 3-SAT (hard clauses are ternary).

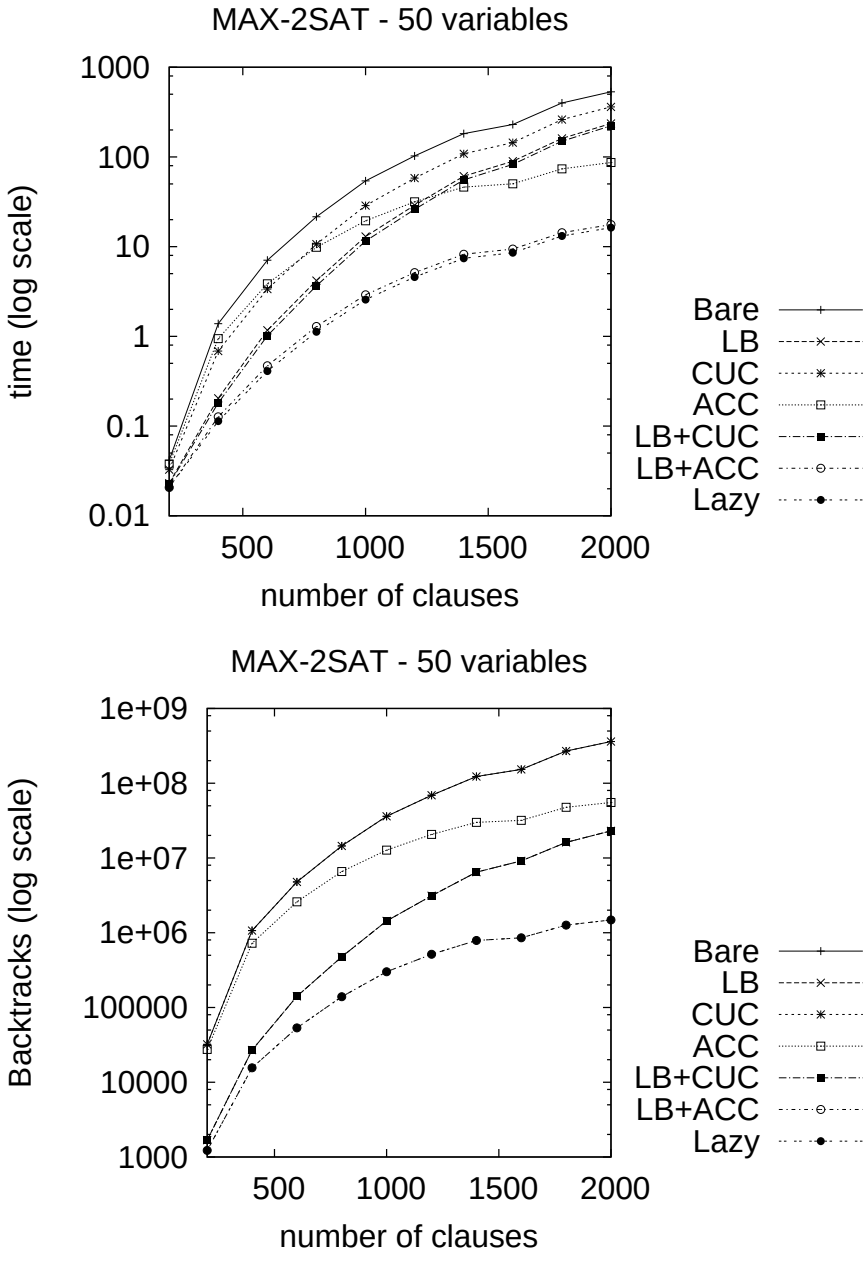


Figure 5.2: Weighted Random MAX-2-SAT 50 variables

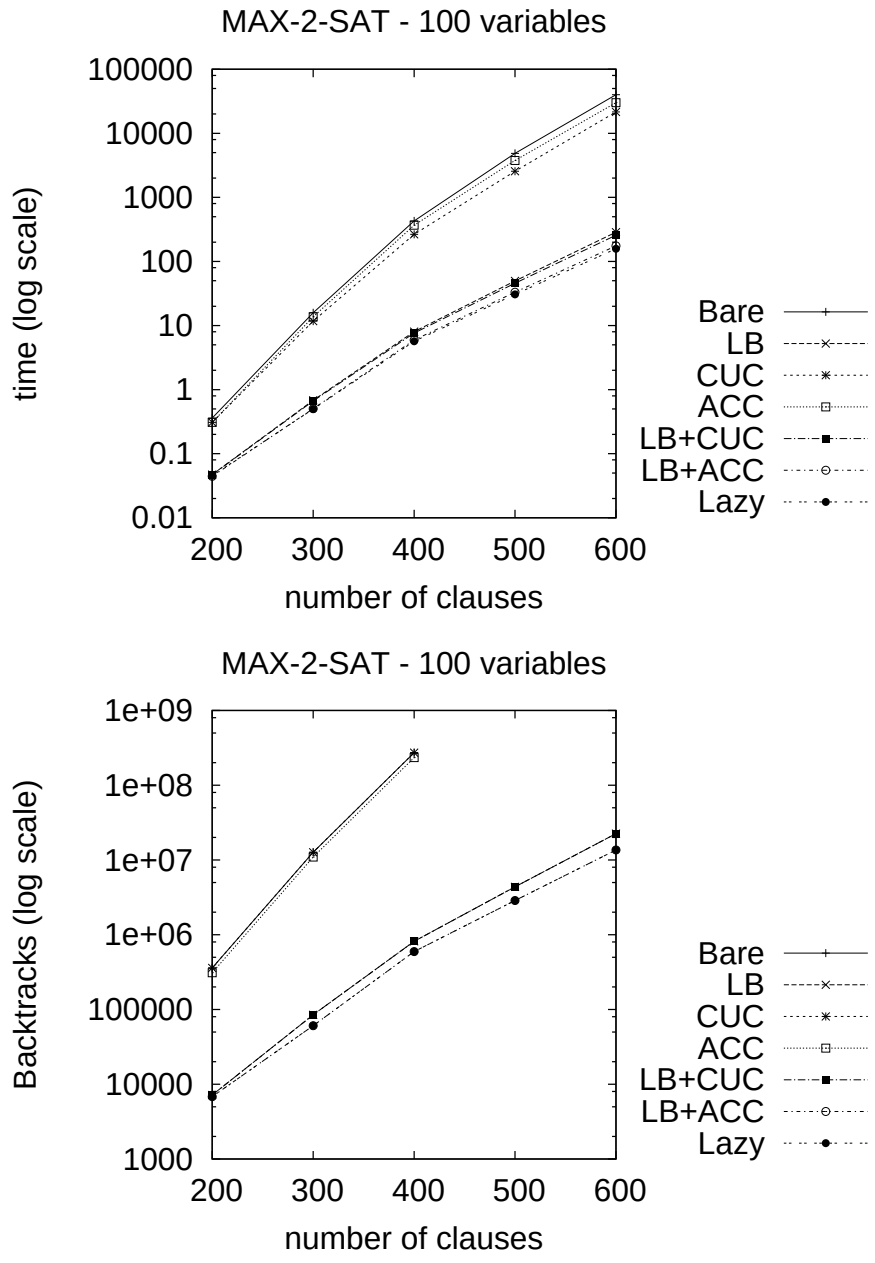


Figure 5.3: Weighted Random MAX-2-SAT 100 variables

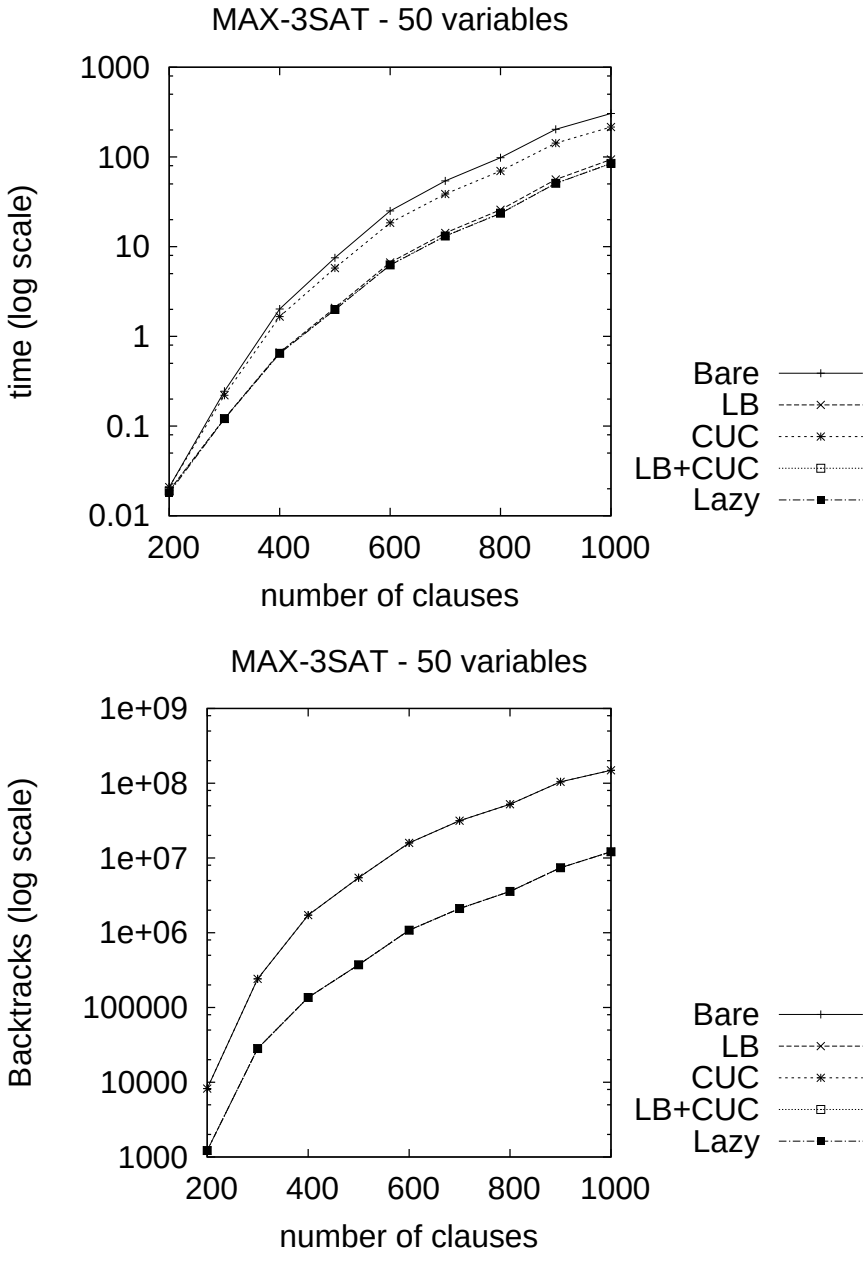


Figure 5.4: Weighted Random MAX-3-SAT 50 variables

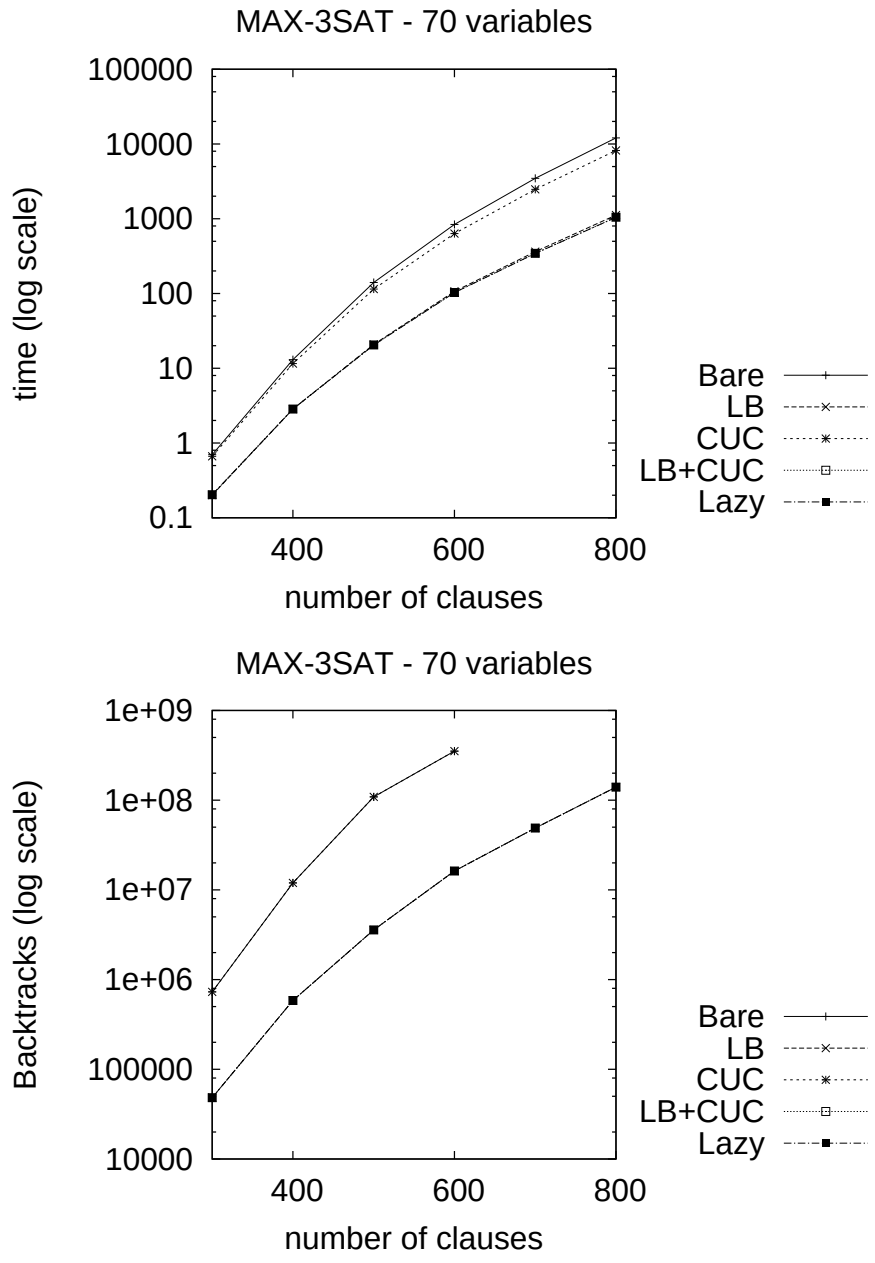


Figure 5.5: Weighted Random MAX-3-SAT 70 variables

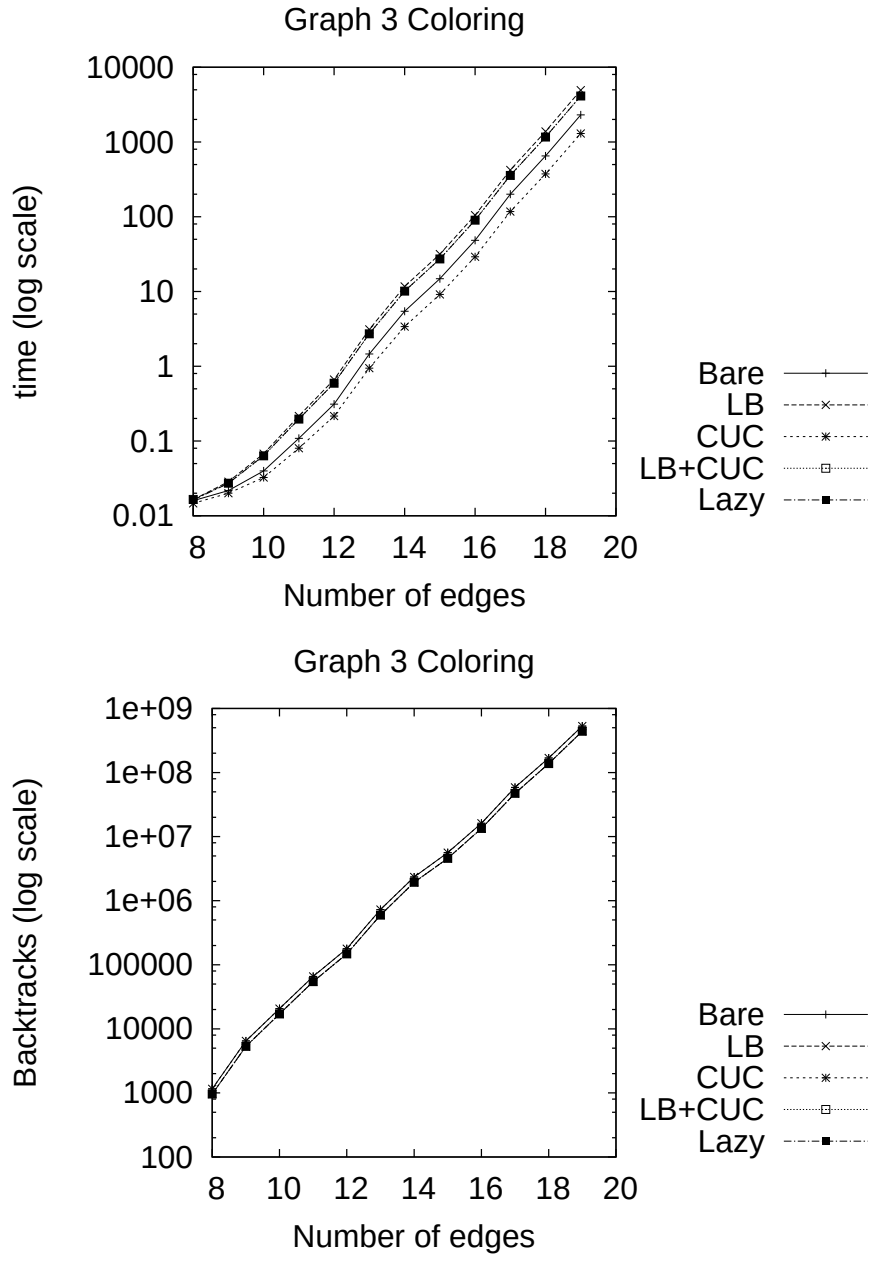


Figure 5.6: Random Graph Coloring

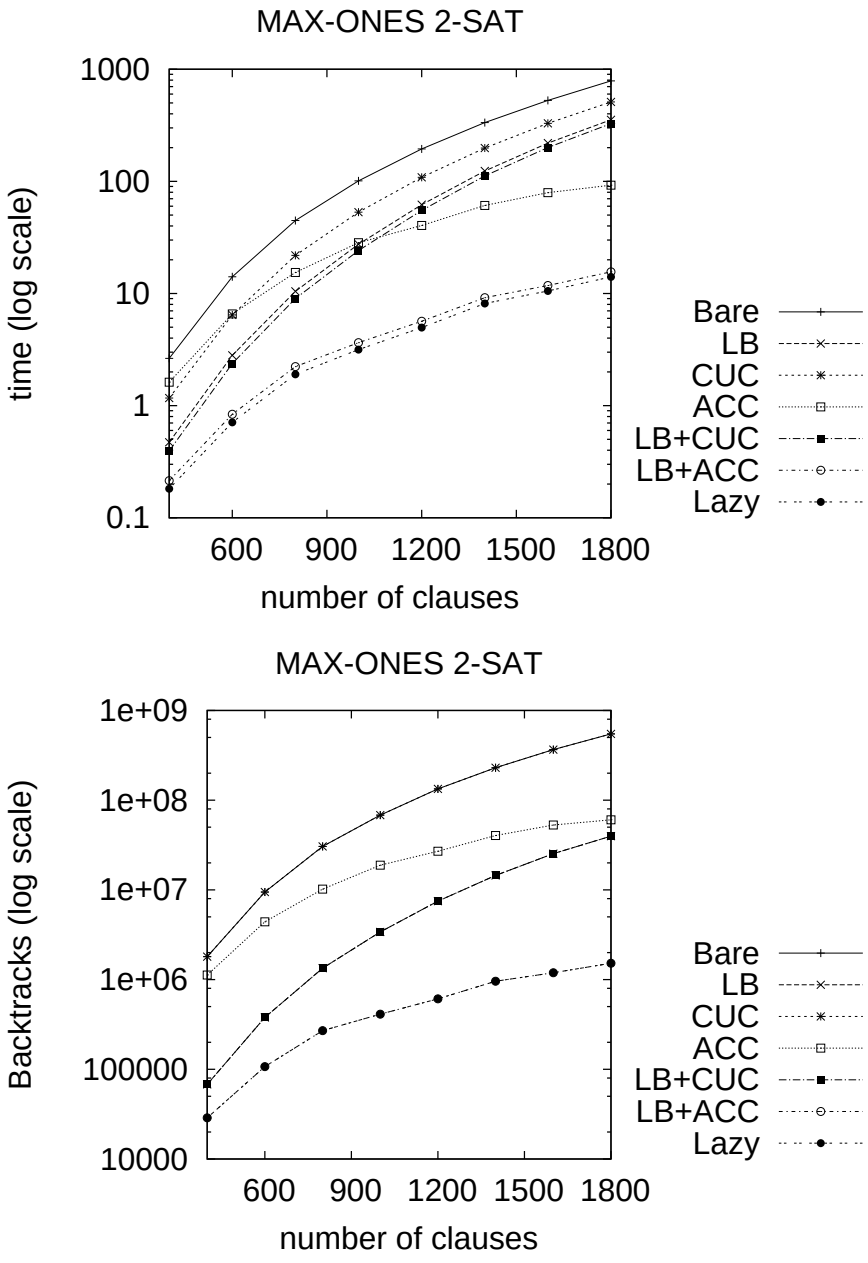


Figure 5.7: Random MAX-ONES 2-SAT

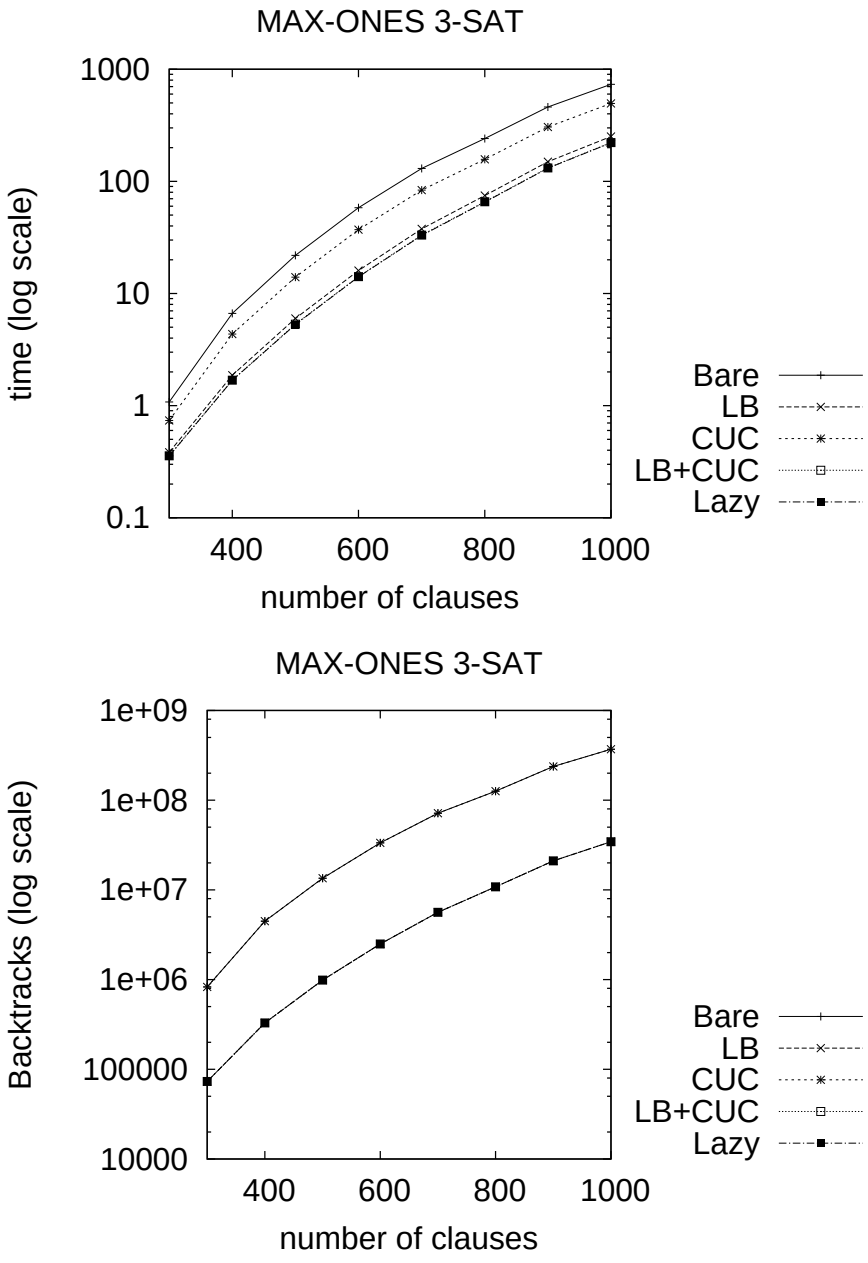


Figure 5.8: Random MAX-ONES 3-SAT

In the fifth experiment, we have run all the weighted instances from the Max-SAT Evaluation 2006 on the seven solvers (cf. Table 5.1). `Lazy` is the best performing, although not in all the sets (it is the best in 14 sets), since `LB+CUC` performs better in 11 sets, and `CUC` in 5 sets.

5.4 Summary

We have defined lazy data structures for a weighted MAX-SAT solver, and a novel variable selection heuristic. Several lower bounds and inference rules have been adapted from MAX-SAT to weighted MAX-SAT, and their performance has been experimentally evaluated. The lower bound and the application of two inference rules, ACC and CUC, improve the results of the solver: the lower bound leads to improvements in all the experiments performed, with more influence in the less constrained region; the performance of the inference rule ACC increases with the number of clauses, then in the most constrained region; and inference rule CUC has a constant and limited impact on the solver performance.

The intuitive reason of the good performance of the inference rules are:

- ACC moves a weight from two binary clauses to a unit clause. This makes possible the application of more lower bound computations, bringing more weight to the empty clause.
- CUC saves time in the computation of the lower bound, although it keeps the number of backtracks.

	Bare	LB	CUC	ACC	LB+CUC	LB+ACC	Lazy
AUCTION PATHS	227.52(11)	283.95(17)	440.21(15)	227.39(11)	236.07(17)	283.42(17)	**85.55(19)**
AUCTION REGIONS	0.27 (30)	0.37 (30)	**0.15 (30)**	0.27 (30)	0.27 (30)	0.37 (30)	2.04 (30)
AUCTION SCHEDULING	21.89(30)	43.73(30)	**9.34 (30)**	21.90(30)	32.15(30)	43.66(30)	63.36(30)
MAXCLIQUE brock	66.66(3)	91.45(3)	229.86(4)	66.71(3)	60.17(3)	91.44(3)	**104.83(4)**
MAXCLIQUE c-fat	0.98 (5)	1.71 (5)	0.62 (5)	1.43 (5)	1.35 (5)	2.74 (5)	**17.57(7)**
MAXCLIQUE hamming	411.13(3)	0.87 (3)	132.98(3)	411.79(3)	338.27(4)	0.86 (3)	**195.08(5)**
MAXCLIQUE johnson	27.97(3)	48.87(3)	**10.56(3)**	27.94(3)	33.52(3)	48.79(3)	38.66(3)
MAXCLIQUE keller	51.07(1)	70.67(1)	**17.62(1)**	51.08(1)	47.91(1)	70.50(1)	43.38(1)
MAXCLIQUE MANN	4.19 (1)	4.12 (1)	3.55 (1)	4.19 (1)	3.98 (1)	4.06 (1)	**0.31 (1)**
MAXCLIQUE p hat	19.83(4)	24.66(4)	6.67 (4)	15.39(3)	16.17(4)	385.94(4)	**216.73(8)**
MAXCLIQUE san	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	**67.84(2)**
MAXCLIQUE sanr	714.56(2)	1100.43(2)	**209.46(2)**	714.01(2)	688.83(2)	1100.44(2)	792.01(2)
WMAXCUT brock	39.48(12)	17.89(12)	28.88(12)	39.70(12)	**16.56(12)**	17.85(12)	18.00(12)
WMAXCUT c-fat	97.08(7)	30.53(7)	64.00(7)	98.08(7)	27.33(7)	30.41(7)	**25.95(7)**
WMAXCUT hamming	385.79(4)	402.62(5)	252.46(4)	386.55(4)	**385.00(5)**	403.72(5)	89.10(4)
WMAXCUT johnson	166.15(3)	65.31(3)	133.47(3)	167.38(3)	**61.86(3)**	64.98(3)	74.44(3)
WMAXCUT keller	41.72(2)	18.04(2)	31.23(2)	41.96(2)	**16.85(2)**	18.01(2)	17.44(2)
WMAXCUT MANN	1274.54(3)	859.36(4)	1057.61(3)	1276.02(3)	**820.49(4)**	856.71(4)	1016.28(4)
WMAXCUT p hat	34.89(12)	11.82(12)	24.47(12)	35.16(12)	**10.91(12)**	11.77(12)	**10.91(12)**
WMAXCUT san	125.85(11)	53.68(11)	97.57(11)	127.70(11)	**50.50(11)**	53.54(11)	57.41(11)
WMAXCUT sanr	63.32(4)	24.32(4)	48.70(4)	64.36(4)	**22.96(4)**	24.19(4)	25.92(4)
WMAXCUT max cut	804.80(13)	303.66(40)	740.35(25)	807.19(13)	266.26(40)	303.05(40)	**247.06(40)**
WMAXCUT SPINGLASS	6.54 (2)	0.19 (2)	3.36 (2)	6.56 (2)	**0.17 (2)**	0.18 (2)	0.26 (2)
MAXONE	485.59(1)	366.36(17)	993.64(2)	484.85(1)	433.64(18)	366.55(17)	**343.58(27)**
QCP	978.29(2)	744.77(1)	844.73(2)	978.80(2)	737.84(1)	742.73(1)	**94.52(6)**
RAMSEY	111.18(28)	37.63(29)	105.21(28)	111.89(28)	**37.22(29)**	37.62(29)	54.87(29)
WCSP DENSE LOOSE	539.61(20)	730.90(21)	510.89(20)	542.33(20)	739.10(21)	729.24(21)	**527.42(32)**
WCSP DENSE TIGHT	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)
WCSP SPARSE LOOSE	465.97(14)	551.02(13)	442.35(14)	467.40(14)	566.68(13)	549.29(13)	**393.11(27)**
WCSP SPARSE TIGHT	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)
WCSP SPOT	0.12 (8)	74.91(9)	0.12 (8)	0.13 (8)	**70.36(9)**	74.76(9)	14.65(6)

Table 5.1: Evaluation results for the seven solvers

Chapter 6

Empirical comparison of MAX-SAT and weighted MAX-SAT solvers

Since Borchers and Furman's work in 1995, there have been many exact solvers addressing MAX-SAT solving. We will compare our contributions to MAX-SAT solving with other researchers' work.

First, we describe all the solvers available to be compared with, then our contributed solvers, and finally we report on the results on MAX-SAT and weighted MAX-SAT, that provide empirical evidence that our best solvers outperform the state-of-the-art solvers on the solved MAX-SAT instances and in many cases on the solved weighted MAX-SAT instances.

6.1 Solvers

In this section we describe ten competitive exact branch and bound solvers for MAX-SAT and weighted MAX-SAT: six were implemented by other researchers and four are our contribution. Since the main contributions of such solvers were given in Section 2.3.3, here we only sketch them, and additionally provide technical details.

6.1.1 Other existing MAX-SAT solvers

Before starting our research on MAX-SAT, there were three implemented solvers:

BF Borchers and Furman [BF95, BF99] implemented branch and bound solvers for MAX-SAT and weighted MAX-SAT. They initialized the initial upper bound using a local search method (cf. Section 2.3) and incorporated the inference rule of unit propagation likewise SAT when the number of unit

clauses is one less than the upper bound (cf. Section 4.1). As variable selection heuristic they used MOMS. The solvers were implemented in C.

Later on, Joy, Mitchell and Borchers [JMB97] compared this solver with a Branch-and-Cut algorithm. This solver is not publicly available.

Wallace&Freuder Wallace and Freuder [WF96] implemented a solver for MAX-SAT. It incorporated lower bound inconsistencies count; and inference rule MAX-SAT-FC. As variable selection heuristic they used MOMS. It was implemented in Lisp. The solver is not publicly available.

AGN For his Master's Thesis, Gramm created a MAX-2-SAT solver [Gra99, GN00], implementing ideas described in [NR00]. He incorporated the following inference rules: pure literal, CUC, DUC, restricted resolution rule, three occurrences rule and ACC. He did not incorporate any underestimation of the lower bound. In order to select the variables, Gramm implemented eight branching rules. Each branching rule depends on the number of occurrences of the literals and on the clauses in which the literals occur. It was implemented in Java.

Since we started our research, there have been a number of researchers that have focused on exact MAX-SAT solver design:

max2sat_lb4a (LB4) Zhang and Shen [ZSM03b, SZ04, SZ05] implemented two solvers for MAX-2-SAT with a static variable selection heuristic, sorting variables by occurrence. Both incorporated a new lower bound computation, LB4, and a new variable selection heuristic based on SCC (cf. Section 2.3.3). The first solver is a decision algorithm. The second one, a MAX-2-SAT solver, applied several preprocessing inference rules. We do not provide results with this solver because we got some non-optimal solutions.[1] The solvers were implemented in C++.

toolbar In CP-2003 [dGLMS03], de Givry et al. introduced a solver that encodes MAX-SAT as a weighted constraint network, which is solved with an algorithm for weighted CSP. In such an algorithm, weighted CSP local consistency is exploited [LS03, LS04, dGZHL05] (version 2 of solver `toolbar`, named `toolbar v2` in the experimentation). The most powerful arc consistency is existential arc consistency.[2] Later, Heras and Larrosa added inference rules focused on MAX-SAT [LH05a, HL06b],[3] adapting the solver to deal with clauses (version 3 of solver `toolbar`, named `toolbar v3` in the experimentation). There is no underestimation implemented. The solver deals with MAX-SAT and weighted MAX-SAT instances. It uses Jeroslow-Wang as variable selection heuristic. It was implemented in C.

[1] The performance of LB4 is similar to AMP.

[2] Many local consistency algorithms in weighted CSP can be seen as inference rules in MAX-SAT, as was stated in [LH05a].

[3] The inference rules added in [LH05a] can be seen as extensions to weighted MAX-SAT of ACC (named NRES in the article), CUC and MAX-SAT-FC. The inference rule DRES in [HL06b] corresponds to the weighted version of Lemma 4.1. This rule was defined independently of our work. The inference rule 2-RES is commented in Appendix A.

MaxSolver In CP-2004, Zhang and Xing [XZ04, XZ05] introduced this solver with a dynamic variable selection heuristic. Actually, `MaxSolver` is a set of four solvers, solving MAX-2-SAT, MAX-3-SAT, weighted MAX-2-SAT and weighted MAX-3-SAT. It uses a new lower bound computation based on integer programming; and the following inference rules: pure literal rule, upper bound rule, DUC rule and coefficient-determining inference rule. Two variants of Jeroslow-Wang variable selection heuristic were used, one for MAX-2-SAT and another one for MAX-3-SAT. The solver is limited to instances with 1000 clauses. It was implemented in C.

Adding up, there are currently five exact MAX-SAT solvers to be compared with: `BF`, `AGN`, `toolbar` and `MaxSolver`. The information is summarized in Table 6.1.

Solver	*Researchers*	*Year of issue*	*Year of last version*	*Weighted*
`BF`	Borchers, Furman	'95	'99	yes
`AGN`	Alber, Gramm, Niedermeier	'99	'00	no
`LB4`	Zhang, Sheng	'03	'05	no
`toolbar`	Givry, Larrosa, Meseguer, Schiex	'03	'06	yes
`MaxSolver`	Xing, Zhang	'04	'05	yes

Table 6.1: MAX-SAT solvers from other research works.

6.1.2 Our contribution

During our research on MAX-SAT solving, we have designed and implemented four solvers:

AMP In SAT-2003 [AMP03a], we introduced this solver, an improvement of solver `BF` by incorporating lower bound inconsistency count and inference rules upper bound rule and DUC. We also introduced in CCIA-2003 [AMP03b] a new variable selection heuristic, a variant of Jeroslow-Wang (cf. Section 2.3.3). Likewise BF solver, these improvements were also implemented in C.

Lazy In IBERAMIA-2004 [AMP04a, AMP04b], we introduced the first solver we have implemented from scratch. The solver implements lower bound star rule for the first time; and inference rules CUC, DUC, and ACC as a preprocessing. It uses a static variable selection heuristic which sorts variables mainly by occurrence. In this solver, we introduced a new variable ordering (cf. Section 5.2.2). We implemented two versions, `Lazy` and `Lazy*`, which have one and two pointers in the clauses, respectively. `Lazy*` was chosen for the experimentation. It was implemented in C++.

UP In CP-2005 [LMP05], we introduced this solver, that incorporates for the first time the lower bound UP. It uses a dynamic variable selection heuristic, the same variant of JW used in AMP. It also implements inference rules CUC, DUC and ACC. It was implemented in C++.

MaxSatz In AAAI-2006 [LMP06], we introduced this solver, created applying lower bound UP^*_{FL} and inference rules defined in Chapter 4, using the data structures in solver `Satz` [LA97a, LA97b]. It also applies the following techniques: pure literal rule, upper bound rule, and DUC, and uses two novel heuristics, one for variable selection and one for value selection, defined below. Let $\mathcal{U}(\ell)$ be the number of unit clauses containing literal ℓ, $\mathcal{B}(\ell)$ be the number of binary clauses containing literal ℓ, $\mathcal{C}(\ell)$ be the number of clauses with three or more literals containing literal ℓ. We defined:

- A novel *variable selection heuristic*: We select the variable p such that $(\mathcal{U}(\neg p) + 4 \times \mathcal{B}(\neg p) + \mathcal{C}(\neg p))^*(\mathcal{U}(p) + 4 \times \mathcal{B}(p) + \mathcal{C}(p))$ is the largest.
- And a novel *value selection heuristic*: Let p be the selected branching variable. If $\mathcal{U}(\neg p) + 4 \times \mathcal{B}(\neg p) + \mathcal{C}(\neg p) < \mathcal{U}(p) + 4 \times \mathcal{B}(p) + \mathcal{C}(p)$, set p to true. Otherwise, set p to false.

The solver was implemented in C.

The solvers we have developed are shown in Table 6.2. `AMP` has not been modified since its creation, but `Lazy`, `UP` and `MaxSatz` have been improved since then.

Solver	*Researchers*	*Year of issue*	*Weighted*
`AMP`	Alsinet, Manyà, Planes	'03	yes
`Lazy`	Alsinet, Manyà, Planes	'04	yes
`UP`	Li, Manyà, Planes	'05	no
`MaxSatz`	Li, Manyà, Planes	'06	no

Table 6.2: MAX-SAT solvers we have implemented

6.2 Experimentation on MAX-SAT

We next report on the experimental investigation of the comparison of the several MAX-SAT solvers. All the experiments were performed on a Linux Cluster with 2GHz AMD Opteron processors with 1Gb of RAM.

We provided the same initial upper bound to all the solvers, which was computed with a GSAT algorithm (cf. Section 2.2.4) for MAX-SAT and weighted MAX-SAT implemented by Borchers and Furman.

In the first experiment, Figure 6.1 and Figure 6.2, we compared the solvers for random MAX-2-SAT instances. We observe that `MaxSatz` outperforms the

rest of the solvers. In MAX-2-SAT with 50 and 100 variables, solver `toolbar` becomes second, although when increasing the number of variables to 150, `UP` is the second one. An important point to stress is the behavior of `MaxSatz` and `toolbar`. `MaxSatz` is faster than `toolbar`, but both join at the more constrained point. The intuition behind is that `toolbar` deals better with repeated clauses than `MaxSatz`. `toolbar` joins repeated clauses into one and labels them with a weight, so that `toolbar` solves a formula with quite fewer clauses than `MaxSatz`, in the most constrained region. Observe also the difference between the two versions of `toolbar` and the two solvers `UP` and `MaxSatz`, that confirms that the more powerful the inference rule, the faster the algorithm (cf. Section 4.4).

In the second experiment, Figure 6.3, we compared the solvers able to solve MAX-3-SAT (i.e., all previous solvers but `AGN`). We observe a similar behavior to random MAX-2-SAT: solver `MaxSatz` is the best performing, `toolbar` has a good performance for instances of 50 variables, and `UP` has for instances of 70 variables.

In the third experiment, Figure 6.4, we compared the solvers for random MAX-CUT instances. We observe that solver `MaxSatz` outperforms the rest of the solvers.

In the fourth experiment, Table 6.3, we compared the solvers on the benchmarks used in the MAX-SAT Evaluation 2006. The first column contains the name of the benchmark set, the second column the number of instances in the set, and the following columns the average time spent by the solver on the solved instances (number in brackets). The maximum time allowed to solve one instance is 30 minutes. Benchmarks not applicable to the solver are marked with a dash. We observe that solver `MaxSatz` outperforms the rest of the solvers.

Solver `MaxSatz` outperforms in all the experiments, and demonstrates to be robust because which solver is in second and third position depends on the problem: `toolbar`, `MaxSolver`, or `UP`.

6.3 Experimentation on weighted MAX-SAT

In this section, we describe the experimentation performed on the five solvers which are able to deal with weighted MAX-SAT: i.e. `BF`, `toolbar`, `MaxSolver`, `AMP` and `Lazy`. In the first experiment, Figure 6.5, we compared the solvers which are able to solve weighted MAX-2-SAT. We observe that solver `toolbar` is the best solver for 50 and 100 variables. `MaxSolver` can solve only instances containing less than 1000 clauses.

In the second experiment, Figure 6.6, we compared the solvers able to solve weighted MAX-3-SAT. We observe that `Lazy` is the best performing solver and `toolbar` is the second one. We observe that the more constrained the problem, the closer the solvers `Lazy` and `toolbar` are. As commented in the previous section regarding the comparison between `toolbar` and `MaxSatz`, we think that this happens because `toolbar` deals better with repeated clauses than `Lazy`.

In the third experiment, Figure 6.7, we compared the solvers able to solve the graph 3-coloring problem. We observe that `Lazy` is the best performing solver

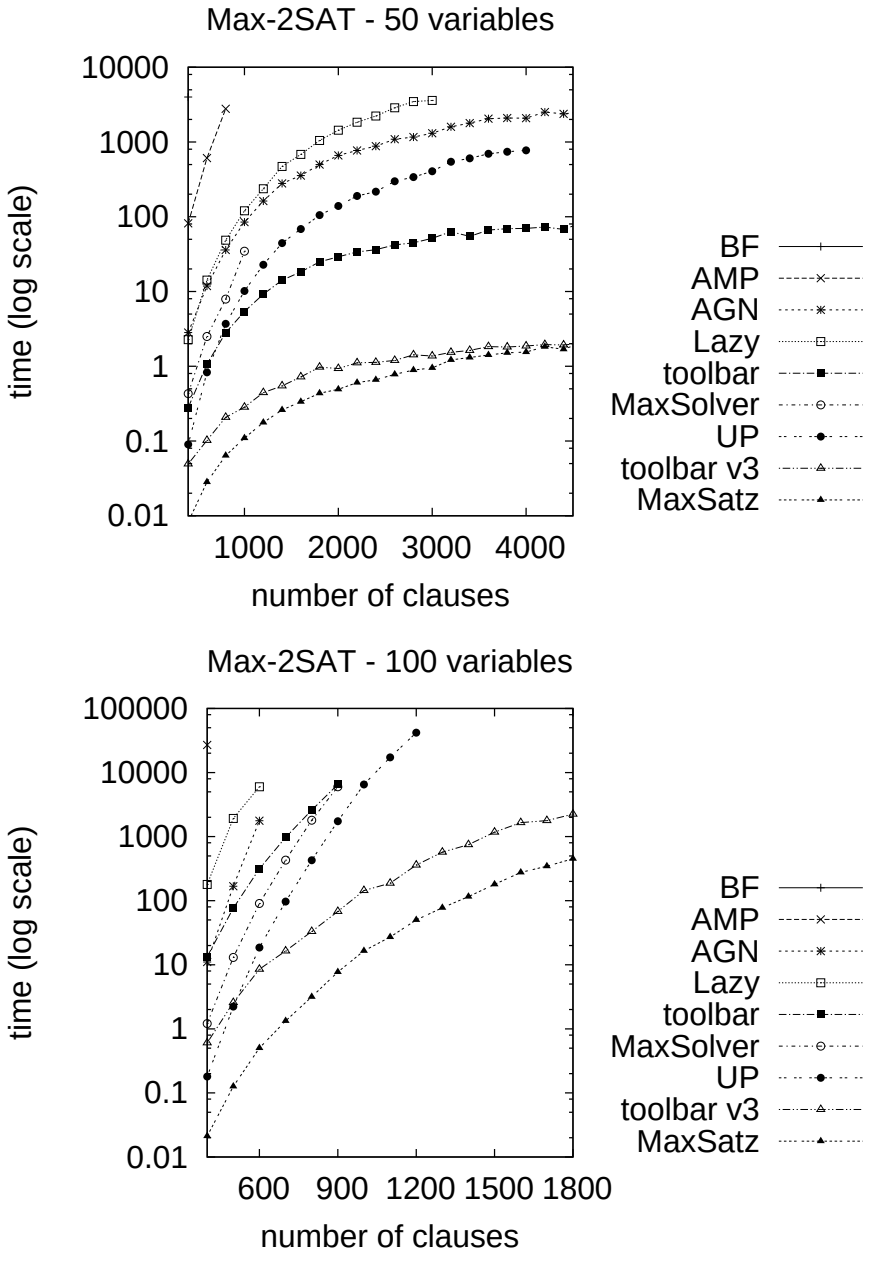


Figure 6.1: Random MAX-2-SAT solver comparison

Set Name	#Instances	BF	AMP	Toolbar v3	Lazy	MaxSolver	UP	MaxSatz
MAXCUT brock	11	(0)	545.81(1)	54.65(11)	152.67(11)	(0)	508.85(8)	**12.59(11)**
MAXCUT c-fat	7	6.06 (1)	1.95 (3)	21.14(5)	132.94(5)	41.58(3)	7.19 (5)	**0.07 (5)**
MAXCUT hamming	6	(0)	636.04(1)	557.24(3)	35.43(2)	(0)	294.89(2)	**180.27(3)**
MAXCUT johnson	4	(0)	394.17(2)	145.59(3)	494.42(3)	1.34 (1)	29.42(2)	**45.38(3)**
MAXCUT keller	2	(0)	197.15(1)	16.97(2)	58.96(2)	(0)	615.54(2)	**6.11 (2)**
MAXCUT DIMACS p hat	12	605.44(2)	107.79(8)	60.79(12)	157.31(12)	14.00(8)	140.23(9)	**15.83(12)**
MAXCUT san	11	(0)	563.19(1)	64.16(7)	227.26(7)	283.34(2)	812.47(5)	**274.88(11)**
MAXCUT sanr	4	(0)	428.18(1)	272.94(4)	443.55(4)	138.32(1)	538.10(3)	**72.00(4)**
MAXCUT max cut	40	0.01 (1)	(0)	34.37(40)	791.83(27)	(0)	623.03(13)	**5.58 (40)**
MAXCUT SPINGLASS	5	0.21 (1)	0.13 (1)	5.63 (2)	44.03(2)	570.68(2)	0.86 (2)	**44.96(3)**
MAXONE	45	0.02 (21)	0.03 (45)	29.47(45)	80.92(40)	0.06 (45)	0.31 (45)	**0.02 (45)**
RAMSEY ram k	48	8.53 (30)	38.44(30)	66.99(28)	76.38(28)	0.20 (20)	19.65(25)	**8.98 (34)**
MAX2SAT 100VARS	50	0.14 (10)	143.23(11)	19.05(50)	228.18(31)	532.47(16)	192.34(48)	**1.40 (50)**
MAX2SAT 140VARS	50	0.08 (10)	91.93(12)	110.81(49)	196.76(23)	168.42(18)	75.57(39)	**7.02 (50)**
MAX2SAT 60VARS	50	1.92 (3)	514.02(44)	0.21 (50)	3.17 (50)	81.82(50)	0.94 (50)	**0.03 (50)**
MAX2SAT DISCARDED	180	357.65(28)	439.54(76)	95.42(174)	135.44(107)	308.58(73)	166.29(149)	**16.80(180)**
MAX3SAT 40VARS	50	170.49(22)	202.18(50)	9.26 (50)	6.89 (50)	66.34(49)	60.50(50)	**1.50 (50)**
MAX3SAT 60VARS	50	4.07 (16)	168.00(25)	317.87(50)	261.87(43)	139.03(22)	166.76(37)	**23.32(50)**

Table 6.3: Experimental results for all the unweighted benchmarks in the MAX-SAT Evaluation 2006.

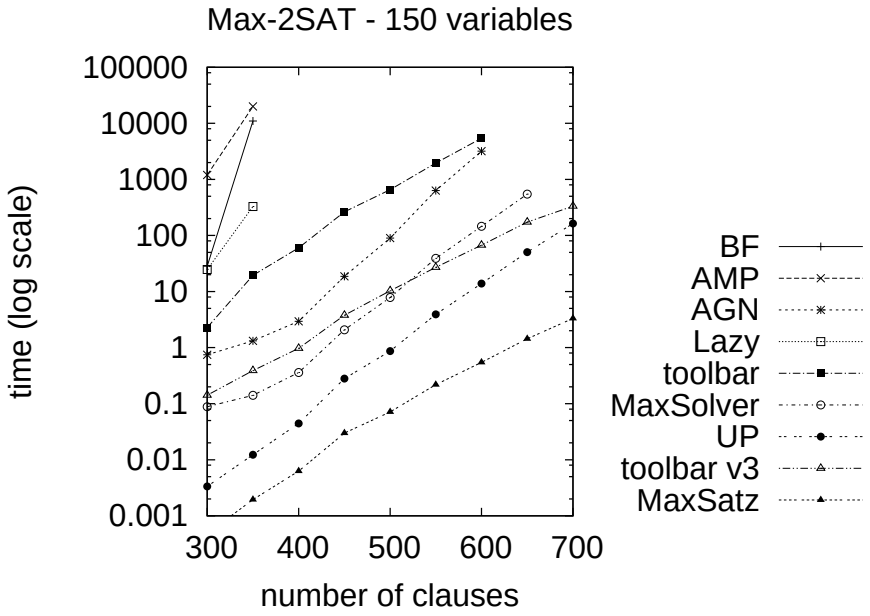


Figure 6.2: Random MAX-2-SAT with 150 variables solver comparison

and `MaxSolver` is the second one. It finishes at 17 nodes due to its limitation of 1000 clauses.

In the fourth experiment, Figure 6.8, we compared the solvers able to solve MAX-ONES. We observe that solver `Lazy` is the best performing solver for MAX-2-SAT and `toolbar` is the best one for MAX-3-SAT. `MaxSolver` shows its limitation of 1000 clauses (in this case, it cannot reach such a limit due to the additional clauses).

In the last experiment, we compared the solvers with all the weighted MAX-SAT benchmarks submitted to the Max-SAT Evaluation 2006. The results are shown in Table 6.4.

We can see that `Lazy` does not outperform in all the experiments, but it does in most of them. In particular, `toolbar` is better for MAX-2-SAT problems, and it is expected to be as powerful as `Lazy` in more constrained MAX-3-SAT problems.

6.4 Summary

We have observed that solver `MaxSatz` outperforms the rest of the solvers for the MAX-SAT instances we have tested. `MaxSatz` is robust because its performance does not depend on the parameters of the problem solved, compared with the other solvers that become second depending on the parameters of the problem.

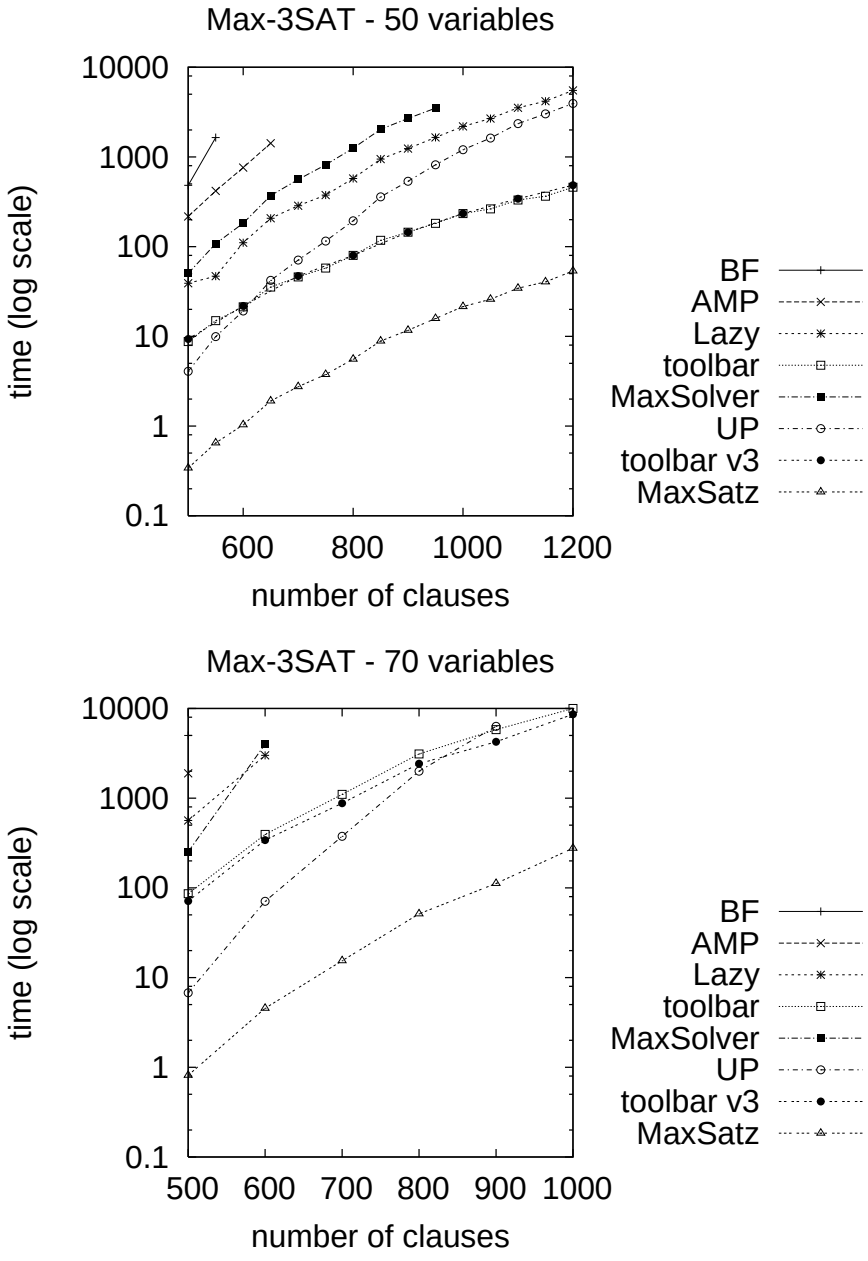


Figure 6.3: Random MAX-3-SAT solver comparison

Set Name	#Instances	BF	AMP	MaxSolver	Lazy	Toolbar v2	Toolbar v3
Auction (paths)	30	416.53(9)	244.32(9)	926.68(6)	85.55(19)	312.27 (17)	**258.32(25)**
Auction (regions)	30	1.53 (1)	0.83 (1)	-	**2.04 (30)**	3.27 (30)	6.07 (30)
Auction (schedulling)	30	10.74(11)	5.85 (11)	-	**63.36(30)**	133.21 (29)	151.88(30)
Max-Clique (brock)	12	(0)	(0)	-	104.83(4)	374.31 (4)	**48.09(8)**
Max-Clique (c-fat)	7	(0)	(0)	-	17.57(7)	6.05 (7)	**10.43(7)**
Max-Clique (hamming)	6	0.35 (2)	0.19 (2)	0.12 (1)	195.08(5)	90.12 (5)	**112.39(6)**
Max-Clique (johnson	4	51.79(3)	28.49(3)	0.21 (2)	**38.66(3)**	75.63 (3)	56.17(3)
Max-Clique (keller)	2	0.00 (0)	(0)	-	43.38(1)	105.59 (1)	**34.79(1)**
Max-Clique (MANN a)	4	3.55 (1)	2.19 (1)	0.70 (1)	0.31 (1)	2.09 (2)	**51.78(3)**
Max-Clique (p hat)	12	(0)	(0)	(0)	216.73(8)	555.62 (4)	**241.95(9)**
Max-Clique (san)	11	(0)	(0)	(0)	67.84(2)	376.42 (1)	**19.61(3)**
Max-Clique (sanr)	4	(0)	(0)	(0)	792.01(2)	458.90 (1)	**847.24(3)**
Weighted Max-Cut (brock)	12	(0)	998.21(2)	1157.07(5)	**18.00(12)**	115.27 (12)	18.57(12)
Weighted Max-Cut (c-fat)	7	32.89(4)	196.63(5)	331.25(7)	25.95(7)	66.25 (7)	**14.48(7)**
Weighted Max-Cut (hamming)	6	(0)	301.09(1)	1457.19(5)	**89.10(4)**	876.18 (4)	146.02(4)
Weighted Max-Cut (johnson)	4	92.46(1)	264.53(2)	0.54 (1)	**74.44(3)**	7.17 (2)	112.01(3)
Weighted Max-Cut (keller)	2	(0)	293.00(1)	-	**17.44(2)**	151.85 (2)	19.97(2)
Weighted Max-Cut (MANN a)	4	(0)	(0)	-	**1016.28(4)**	(0)	1473.46(1)
Weighted Max-Cut (p hat)	12	528.49(3)	146.10(8)	9.76 (8)	**10.91(12)**	93.66 (12)	19.92(12)
Weighted Max-Cut (san)	11	(0)	742.85(2)	1238.34(6)	**57.41(11)**	361.10 (10)	113.02(11)
Weighted Max-Cut (sanr)	4	(0)	289.52(1)	54.29(1)	**25.92(4)**	223.00 (4)	78.90(4)
Weighted Max-Cut (random)	40	(0)	(0)	-	247.06(40)	544.16 (38)	**17.52(40)**
Weighted Max-Cut (spinglass)	5	(0)	(0)	611.55(3)	0.26 (2)	0.41 (2)	**91.85(3)**
Max-One	45	1042.21(1)	1217.12(3)	117.88(4)	343.58(27)	556.21 (37	**186.15(44)**
Quasigroup Completion	25	2.38 (10)	1.14 (10)	-	94.52(6)	183.19 (6)	**36.35(10)**
Ramsey	48	3.65 (31)	1.94 (31)	46.53(35)	54.87(29)	39.19 (30)	**7.16 (35)**
Weighted CSP (DENSE LOOSE)	40	98.19(39)	57.19(39)	54.58(16)	527.42(32)	145.74 (22)	**238.69(34)**
Weighted CSP (DENSE TIGHT)	60	(0)	(0)	1619.47(10)	(0)	422.84 (30)	**68.61(30)**
Weighted CSP (SPARSE LOOSE)	40	57.98(40)	32.78(40)	2.10(11)	393.11(27)	124.41 (27)	**156.03(36)**
Weighted CSP (SPARSE TIGHT)	40	(0)	(0)	-	(0)	1304.64 (1)	**298.57(20)**
Weighted CSP (spot)	42	153.38(4)	73.62(4)	-	14.65(6)	150.40 (13)	**78.89(16)**

Table 6.4: Experimental results for all the weighted benchmarks in the MAX-SAT Evaluation 2006.

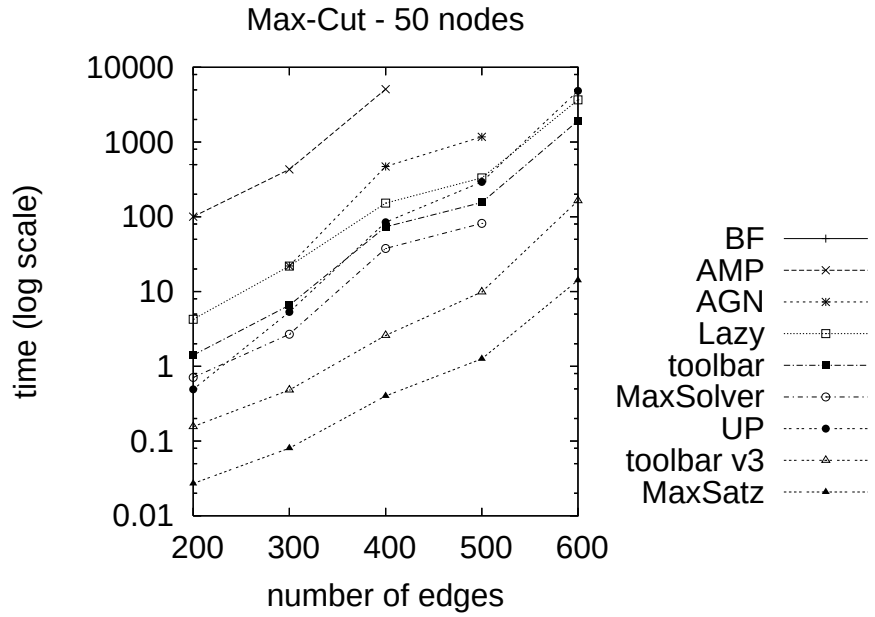


Figure 6.4: Random MAX-CUT solver comparison

Solver `toolbar` performs well when the clause to variable ratio becomes large, and other solvers like `UP` perform better when such a ratio is small.

In the weighted MAX-SAT experimentation, solver `Lazy` has a good but limited performance. The designed lazy data structures make difficult to efficiently implement more complex techniques like lower bound UP or weighted versions of the inference rules defined in Chapter 4. On the other hand, `toolbar` incorporates powerful inference rules, that make the solver have a good performance when the clause to variable ratio is large in random instances.

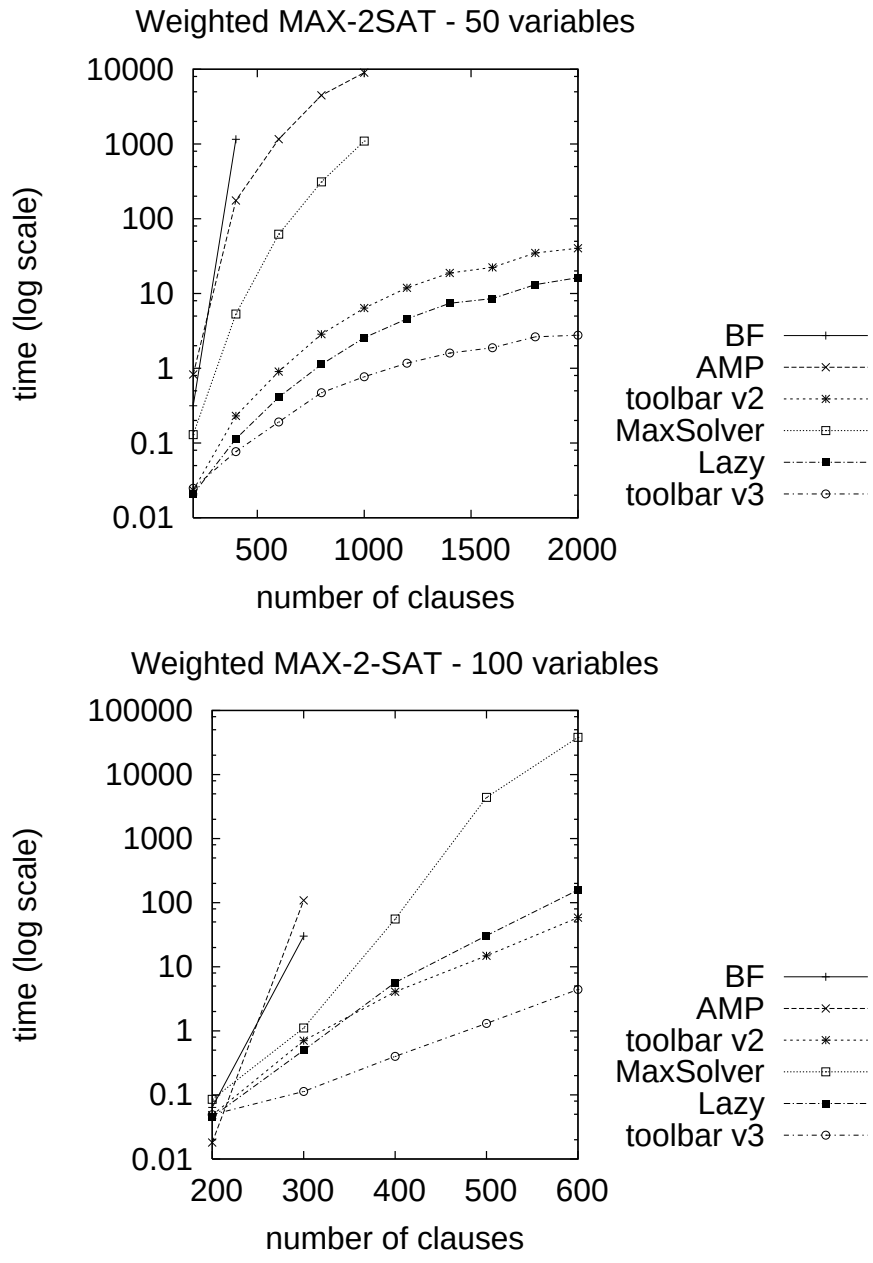


Figure 6.5: Random weighted MAX-2-SAT solver comparison

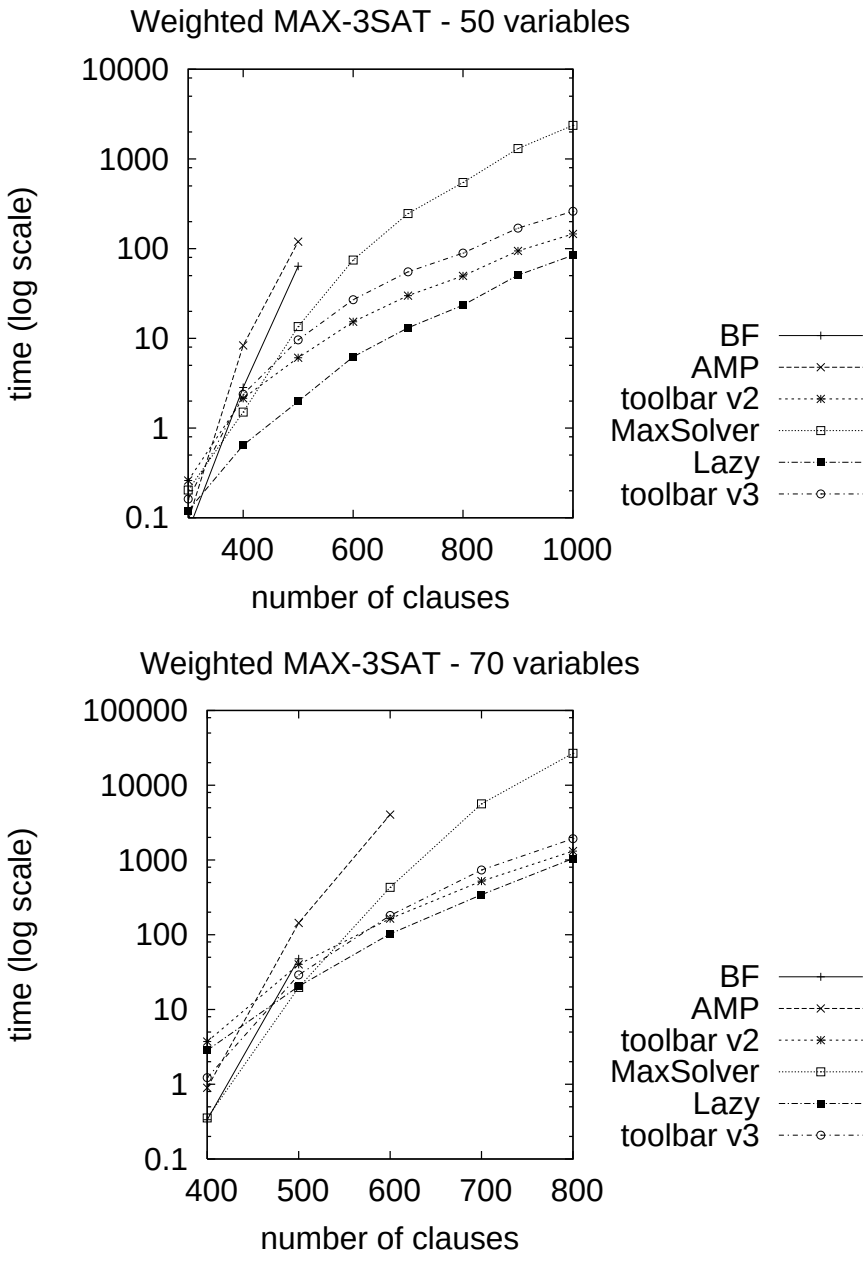


Figure 6.6: Random weighted MAX-3-SAT solver comparison

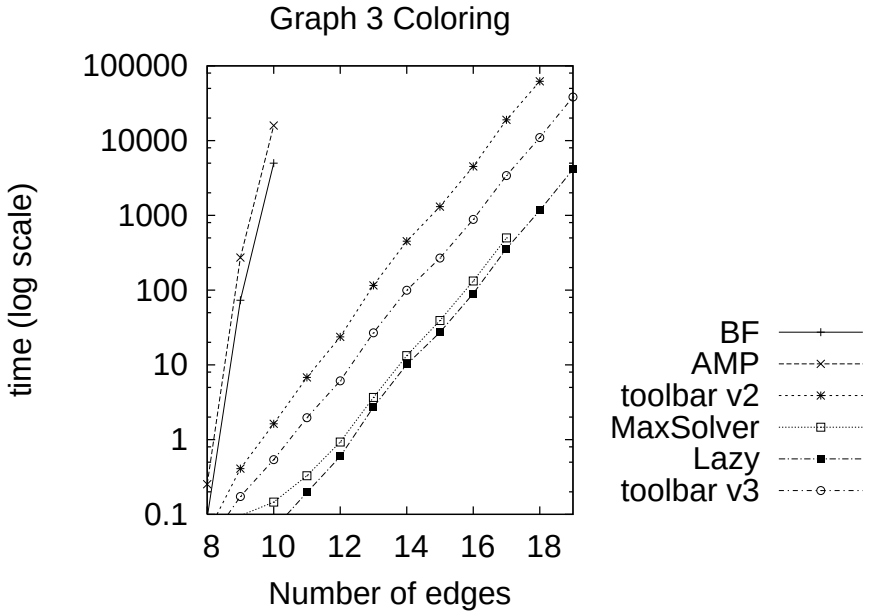


Figure 6.7: Graph coloring solver comparison

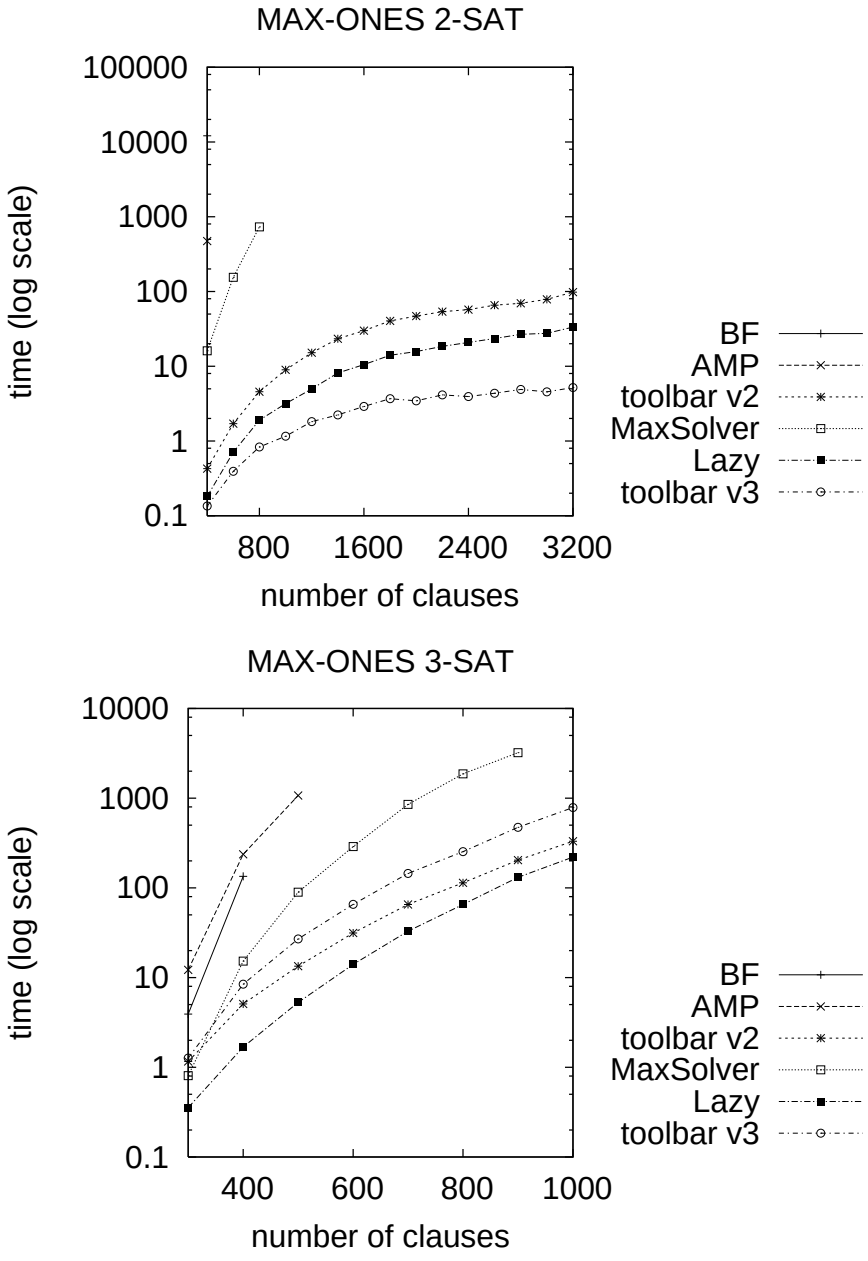


Figure 6.8: MAX-ONES solver comparison

Chapter 7

Conclusions

In recent years, research on MAX-SAT solving has reduced in many orders of magnitude the required time to solve a large number of MAX-SAT instances. In this dissertation, we have focused on two important features that have a great impact on the solvers performance: the lower bound computation and the inference rules. We have argued that unit propagation in MAX-SAT can be used to compute a good lower bound that sustains the application of inference rules that accelerate the search.

The main contributions of this research on those aspects can be summarized as follows:

- Unit propagation, one of the most useful methods in SAT, is extensively used in the computation of a good quality *lower bound*, that detects disjoint inconsistent subsets of clauses. We improved such a lower bound with choosing the appropriate heuristic for selecting unit clauses. The goal of the better heuristic is to use the minimum number of initial unit clauses to detect small inconsistent subsets.

- When there is no unit clause to apply the lower bound, or the existing unit clauses do not lead to any conflict, an estimation can be made detecting failed literals.

- Given an inconsistent subset of clauses detected by the lower bound in the search tree, the same subset has probably to be detected again in downward nodes. Transforming such a subset, using *inference rules*, into an equivalent subset with an empty clause, makes the algorithm run faster, because it avoids to detect the subset again, and the new added clauses may help to detect more inconsistent subsets.

- The key point of the efficient application of the introduced inference rules is the use of the *implication graph* created by the lower bound UP when detecting empty clauses. This fact and the data structures used (from solver `Satz`) bring the creation of an extremely competitive MAX-SAT

solver, `MaxSatz`, that was the best performing solver in the MAX-SAT Evaluation 2006.

- We have provided empirical evidence that the solver `MaxSatz`, that implements the previous points, outperforms the rest of the solvers on the MAX-SAT instances we have tested. `MaxSatz` is robust because its performance does not depend on the parameters of the problem solved, compared with the other solvers that become second depending on such parameters.

- We have defined a lazy data structure and a novel variable selection heuristic in a weighted MAX-SAT solver, called `Lazy`. A lower bound and two inference rules have been adapted from MAX-SAT to weighted MAX-SAT, and their performance has been experimentally evaluated.

- In the weighted MAX-SAT experimentation, solver `Lazy` has a good but limited performance. The designed lazy data structure makes difficult to efficiently implement more complex rules, but it helps the algorithm to run faster.

There are many extensions to the current work, but we consider that the feasible points to be exploited in the near future are:

- Learning is used in SAT to solve structured problems faster [MMZ$^+$01]. This is a topic not sufficiently exploited in MAX-SAT. We believe that it is worth to continue exploring the work in [AM06b].

- The inference rules applied in `MaxSatz` were a subset of all the possible rules, because we forced them not to add new literals to the formula. More powerful inference rules, like the ones in Appendix A, could be applied adapting the solver to the use of dynamic memory.

- We have demonstrated the good performance of lower bounds and inference rules based on unit propagation in MAX-SAT. We think such results can be transferred into weighted MAX-SAT to get also a powerful solver.

- Bistarelli and O'Sullivan [BO04, SBO07] pointed out a branch and bound algorithm with symmetry breaking for soft CSP problems. These ideas could be ported to the MAX-SAT branch and bound algorithm.

- The lower bound may detect the same unsatisfiable set of clauses again and again. In order to save time, incremental lower bound computation could be incorporated.

Appendix A

Additional inference rules

In this chapter we summarize inference rules that, we believe, could improve the performance of MAX-SAT solvers and weighted MAX-SAT solvers. We did not implemented them because the rules need the solver to be able to deal with dynamic memory management. This set of rules increments the number of unit clauses in the formula.

A.1 Unit clause creation rules

While static memory management is sufficient to implement the introduced rules in previous chapters, the implementation of the following rules needs dynamic memory management. In this case, the transformed formula is bigger than the original one, compared with what happens in Chapter 4.

The inference rules that introduce a new empty clause in a CNF formula can be transformed into rules that introduce unit clauses into the formula. These rules can be applied once failed literal detection (cf. Section 3.4.2) has found a contradiction for one of the literals and not for its complementary.

We present two rules. The transformation for Rule 4.4 is as follows:

Rule A.1 *If* $\phi_1 = \{ \bar{l}_1 \vee l_2,\ \bar{l}_2 \vee l_3,\ \ldots,\ \bar{l}_k \vee l_{k+1},\ \bar{l}_{k+1}\} \cup \phi'$, $\phi_2 = \{\bar{l}_1,\ l_1 \vee \bar{l}_2,\ l_2 \vee \bar{l}_3,\ \ldots,\ l_k \vee \bar{l}_{k+1}\} \cup \phi'$, *then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof Adding the literal l_1 to ϕ_1 we can apply Rule 4.4, obtaining a formula with an empty clause. Then, we can replace the empty clause by the complementary unit clauses $\{l_1, \bar{l}_1\}$. Finally, if we remove l_1 from both sides we obtain ϕ_2. ■

This is the application of Lemma 4.1 in a linear derivation, that was introduced as Lemma 2 in [Yan94]. We also present the transformation for Rule 4.5:

Rule A.2 *If* $\phi_1 = \{l_1 \vee l_2,\ l_1 \vee l_3,\ \bar{l}_2 \vee \bar{l}_3\} \cup \phi'$ *and* $\phi_2 = \{\bar{l}_1,\ \bar{l}_1 \vee \bar{l}_2 \vee \bar{l}_3,\ l_1 \vee l_2 \vee l_3\} \cup \phi'$, *then* ϕ_1 *and* ϕ_2 *are equivalent.*

That rule transforms a MAX-2-SAT problem into a MAX-3-SAT problem with a unit clause. This rule can be seen as an extension of Rule 4.1 (ACC), a useful rule because it creates unit clauses from binary clauses. Although Rule A.1 increases the size of the problem, the behavior of ACC makes us think it may speed up the search due the the additional unit clause. Rule A.1 was originally stated for weighted MAX-SAT in [HL06b], and named Hyper Resolution (cf. Section 4.1).

Rule 4.6 was introduced as an addition of a linear derivation in the left side of literal l_{k+1} in Rule 4.5. A linear derivation in the right side of that literal can also be introduced, as follows:

Rule A.3 *If* $\phi_1=\{l_1,\ \bar{l}_1\vee l_2,\ \bar{l}_2\vee l_3,\ldots,\ \bar{l}_k\vee l_{k+1},\ \bar{l}_{k+1}\vee\bar{l}_1\}\cup\phi'$ *and* $\phi_2=\{\square,\ l_1\vee\bar{l}_2\vee l_3,\ldots,\ l_1\vee\bar{l}_k\vee l_{k+1},\ \ \bar{l}_1\vee l_2\vee\bar{l}_3,\ldots,\ \bar{l}_1\vee l_k\vee\bar{l}_{k+1}\}\cup\phi'$ *then* ϕ_1 *and* ϕ_2 *are equivalent.*

Proof Taking literal l_1, we have two possibilities:

- l_1 is satisfied, then ϕ_1 becomes $\{l_2,\ \bar{l}_2\vee l_3,\ldots,\ \bar{l}_k\vee l_{k+1},\ \bar{l}_{k+1}\}\cup\phi'$, and applying Rule 4.2 becomes $\{\square, l_2\vee\bar{l}_3,\ldots,\ l_k\vee\bar{l}_{k+1}\}\cup\phi'$.
- l_1 is not satisfied, then ϕ_1 becomes $\{\square,\ \bar{l}_2\vee l_3,\ldots,\ \bar{l}_k\vee l_{k+1}\}\cup\phi'$.

Unifying both formulas, we obtain ϕ_2. ■

We can also extend this rule to the creation of unit clauses:

Rule A.4 *If* $\phi_1=\{\bar{l}_1\vee l_2,\ \bar{l}_2\vee l_3,\ldots,\ \bar{l}_k\vee l_{k+1},\ \bar{l}_{k+1}\vee\bar{l}_1\}\cup\phi'$ *and* $\phi_2=\{\bar{l}_1,\ l_1\vee\bar{l}_2\vee l_3,\ldots,\ l_1\vee\bar{l}_k\vee l_{k+1},\ \ \bar{l}_1\vee l_2\vee\bar{l}_3,\ldots,\ \bar{l}_1\vee l_k\vee\bar{l}_{k+1}\}\cup\phi'$ *then* ϕ_1 *and* ϕ_2 *are equivalent.*

A corollary of this rule is applied in the SCC lower bound computation [SZ05]. The corollary is as follows:

Lemma A.1 *If* $\phi_1=\{\bar{l}_1\vee l_2,\ \bar{l}_2\vee l_3,\ldots,\ \bar{l}_k\vee l_{k+1},\ \bar{l}_{k+1}\vee\bar{l}_1\}\cup\phi'$*, then* ϕ_1 *can be underestimated by* $\bar{l}_1\cup\phi'$.

Bibliography

[ABLM07] Carlos Ansótegui, María Luisa Bonet, Jordi Levy, and Felip Manyà. The logic behind weighted CSP. In M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), Hyderabad, India*, pages 32–37, 2007.

[ADM+06] Josep Argelich, Xavier Domingo, Felip Manyà, Jordi Planes, and Chu Min Li. Towards solving many-valued MaxSAT. In T. Uemura, editor, *Proceedings of the 36th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2006)*, Singapore, 2006. IEEE Computer Society. Article 26.

[AGKS00] Dimitris Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. Generating satisfiable problem instances. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000), Austin/TX, USA*, pages 256–261. AAAI Press, 2000.

[AGN01] Jocher Alber, Jens Gramm, and Rolf Niedermeier. Faster exact algorithms for hard problems: A parameterized point of view. *Discrete Mathematics*, 229(1–3):3–27, 2001.

[AJ03] Derek E. Armstrong and Sheldon H. Jacobson. Studying the complexity of global verification for NP-Hard discrete optimization problems. *Journal of Global Optimization*, 27:83–96, 2003.

[ALMP08] Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. The first and second Max-SAT evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:251–278, 2008.

[AM03] Carlos Ansótegui and Felip Manyà. Una introducción a los algoritmos de satisfactibilidad. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 20:43–56, 2003.

[AM06a] Josep Argelich and Felip Manyà. Exact Max-SAT solvers for over-constrained problems. *Journal of Heuristics*, 12(4–5):375–392, 2006.

[AM06b] Josep Argelich and Felip Manyà. Learning hard constraints in Max-SAT. In *Proceedings of the Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2006)*, pages 5–12, Caparica, Portugal, 2006.

[AMP03a] Teresa Alsinet, Felip Manyà, and Jordi Planes. Improved branch and bound algorithms for Max-2-SAT and weighted Max-2-SAT. In *Proceedings of the 6th Catalan Conference on Artificial Intelligence (CCIA 2003)*, volume 100 of *Frontiers in Artificial Intelligence and Applications*, pages 435–442, P. Mallorca, Spain, 2003. IOS Press.

[AMP03b] Teresa Alsinet, Felip Manyà, and Jordi Planes. Improved branch and bound algorithms for Max-SAT. In *Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing (SAT 2003)*, pages 408–415, Portofino, Italy, 2003.

[AMP04a] Teresa Alsinet, Felip Manyà, and Jordi Planes. A Max-SAT solver with lazy data structures. In *Proceedings of the 9th Ibero-American Conference on Artificial Intelligence (IBERAMIA 2004)*, volume 3315 of *LNAI*, pages 334–342, Puebla, México, 2004. Springer.

[AMP04b] Teresa Alsinet, Felip Manyà, and Jordi Planes. A Max-SAT solver with lazy data structures. In L.T. Hoai An and P.D. Tao, editors, *Modelling, Computation and Optimization in Information Systems and Management Sciences (MCO 2004)*, pages 491–498, Metz, France, 2004. Hermes publishing.

[AMP05] Teresa Alsinet, Felip Manyà, and Jordi Planes. Improved exact solver for weighted Max-SAT. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *LNCS*, pages 371–377, St. Andrews, Scotland, 2005. Springer.

[Anj05] Miguel F. Anjos. Semidefinite optimization approaches for satisfiability and maximu-satisfiability problems. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:1–47, 2005.

[ASM06] Fadi Aloul, Karem Sakallah, and Igor Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(2):549–558, 2006.

[BF95] Brian Borchers and Judith Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. Technical report, Mathematics Department, New Mexico Institute of Mining and Technology, October 1995.

[BF99] Brian Borchers and Judith Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2:299–306, 1999.

[BGS99] Laure Brisoux, Éric Grégoire, and Lakhdar Saïs. Improving backtrack search for SAT by means of redundancy. In *Proceedings of Foundations of Intelligent Systems, 11th International Symposium (ISMIS 1999)*, pages 301–309, Warsaw, Poland, 1999.

[BHZ06] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38(4), 2006. Art. 12.

[BKS04] Paul Beam, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[BLM06] María Luisa Bonet, Jordi Levy, and Felip Manyà. A complete calculus for Max-SAT. In A. Biere and C. Gomes, editors, *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, volume 4121 of *LNCS*, pages 240–251. Springer, 2006.

[BM00] Ramón Béjar and Felip Manyà. Solving the round robin problem using propositional logic. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000)*, pages 262–266, Austin/TX, USA, 2000. AAAI Press.

[bM05] Mohamed El bachir Menai. A two-phase backbone-based search heuristic for partial MAX-SAT. In M. Ali and F. Esposito, editors, *Proceedings of 18th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2005)*, volume 3533 of *LNCS*, pages 681–684, Bari, Italy, 2005. Springer.

[BMS00] Luís Baptista and João Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In R. Dechter, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *LNCS*, pages 489–494. Springer, 2000.

[BMZ05] Alfredo Braunstein, Marc Mezard, and Riccardo Zecchina. Survey propagation: an algorithm for satisfiability. *Random Structures and Algorithms*, 27:201–226, 2005.

[BO04] Stefano Bistarelli and Barry O'Sullivan. Combining branch & bound and SBDD to solve soft CSPs. In W. Harvey and Z. Kiziltan, editors, *Proceedings of the 4th International Workshop on Symmetry and Constraint Satisfaction Problems (SymCon 2004)*, pages 9–17, Toronto, Canada, 2004.

[BR99] Nikhil Bansal and Venkatesh Raman. Upper bounds for MaxSat: Further improved. In *Proceedings of the 10th International Symposium on Algorithms and Computation (ISAAC 1999)*, volume 1741 of *LNCS*, pages 247–260, Chennai, India, 1999. Springer.

[BS94] Belaid Benhamou and Lakhdar Saïs. Tractability through symmetries in propositional calculus. *Journal of Automatic Reasoning*, 12(1):89–102, 1994.

[BS97] Roberto J. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997)*, pages 203–208, Providence/RI, USA, 1997. AAAI Press.

[BS03] Daniel Le Berre and Laurent Simon. The essentials of the SAT'03 competition. In A. Tacchella and E. Giunchiglia, editors, *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 452–467. Springer, 2003.

[BS06] Daniel Le Berre and Laurent Simon, editors. *Journal on Satisfiability, Boolean Modeling and Computation*, volume 2, chapter Special Volume on the SAT 2005 competitions and evaluations. IOS Press, 2006.

[BS07] Belaid Benhamou and Mohamed Réda Saïdi. Local symmetry breaking during search in CSPs. In F. Bacchus and M. L. Ginsberg, editors, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *LNCS*, pages 195–209, Providence/RI, USA, 2007. Springer.

[CA93] James Crawford and Larry Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI 1993)*, pages 21–27, Washington/DC, USA, 1993. AAAI Press.

[CA96] James Crawford and Larry Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81:31–57, 1996.

[CGLR96] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR 1996)*, Cambridge/MA, USA, 1996. Morgan Kaufmann.

[CIKM97] Byungki Cha, Kazuo Iwama, Yahiko Kambayashi, and Shuichi Miyazaki. Local search algorithms for partial MAXSAT. In *Proceedings of the 14th National Conference on Artificial Intelligence*

(AAAI 1997), pages 263–268, Providence/RI, USA, 1997. AAAI Press.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[CM97] Stephen Cook and David G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In D.Z. Du, J. Gu, and P. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.

[Coo71] Stephen Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, pages 151–158. ACM, 1971.

[CS00] Philippe Chatalic and Laurent Simon. ZRes: The old Davis-Putnam procedure meets ZBDD. In D. McAllester, editor, *Proceedings of 17th International Conference on Automated Deduction (CADE 2000)*, volume 1831 of *LNCS*, pages 449–454. Springer, 2000.

[DABC93] Olivier Dubois, Pascal André, Yacine Boufkhad, and Jaques Carlier. Can a very simple algorithm be efficient for solving SAT problem? In *Proceedings of the DIMACS Challenge II Workshop*, 1993.

[DD01] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 248–253, Seattle/WA, USA, 2001. Morgan Kaufmann.

[dGLMS03] Simon de Givry, Javier Larrosa, Pedro Meseguer, and Thomas Schiex. Solving Max-SAT as weighted CSP. In *9th International Conference on Principles and Practice of Constraint Programming (CP 2003), Kinsale, Ireland*, volume 2833 of *LNCS*, pages 363–376. Springer, 2003.

[dGZHL05] Simon de Givry, Matthias Zytnicki, Federico Heras, and Javier Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 84–89, Edinburgh, Scotland, 2005. Morgan Kaufmann.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[FL05] Jin-Kao Hao Frédéric Lardeux, Frédéric Saubion. Three truth values for the SAT and MAX-SAT problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 187–192, Edinburgh, Scotland, 2005. Morgan Kaufmann.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In A. Biere and C. Gomes, editors, *Proceedings of the 9th International Conference on the Theory and Applications of Satisfiability Testing (SAT 2006)*, volume 4121 of *LNCS*, pages 252–265, Seattle/WA, USA, 2006. Springer.

[FP83] John Franco and Marvin Paull. Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.

[FR04] Hai Fang and Wheeler Ruml. Complete local search for propositional satisfiability. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004)*, pages 161–166, San Jose, California, 2004. AAAI Press.

[Fre95] Jon William Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[GEI91] Malik Ghallab and Gonzalo Escalada-Imaz. A linear control algorithm for a class of rule-based systems. *Journal of Logic Programming*, 11:117–132, 1991.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.

[GKSS07] Carla Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. *Handbook of Knowledge Representation*, chapter Satisfiability Solvers. Foundations of Artificial Intelligence. Elsevier, 2007.

[GN00] Jens Gramm and Rolf Niedermeier. Faster exact solutions for Max2Sat. In G. Bongiovanni, G. Gambosi, and R. Petreschi, editors, *Proceedings of the 4th Italian Conference on Algorithms and Complexity (CIAC 2000)*, volume 1767 of *LNCS*, pages 174–186. Springer, 2000.

[GN01] Evgueni Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT solver. In *Proceedings of Design, Automation and Test in Europe (DATE 2002)*, pages 142–149, Paris, France, 2001. IEEE Computer Society.

[GPFW97] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In D.Z. Du, J. Gu, and P. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.

[Gra99] Jens Gramm. Exact algorithms for Max2Sat and their applications. Master's thesis, Universität Tübingen, 1999.

[GSCK00] Carla Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[GSK98] Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998)*, pages 431–437, Madison/WI, USA, 1998. AAAI Press.

[GvHL06] Carla Gomes, Willem-Jan van Hoeve, and Lucian Leahu. The power of semidefinite programming relaxations for MAXSAT. In C. Beck and B. Smith, editors, *Proceedings of the Third International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2006)*, volume 3990 of *LNCS*, pages 104–118, Cork, Ireland, 2006. Springer.

[GW93a] Ian Gent and Toby Walsh. Easy problems are sometimes hard. *Artificial Intelligence Research*, 1:23–57, 1993.

[GW93b] Ian Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI 1993)*, pages 28–33, Washington/DC, USA, 1993. AAAI Press.

[GW95] Michael Goemans and David Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1145, 1995.

[HC97] Wen Qi Huang and Jin Ren Chao. Solar: A learning from human algorithm for solving SAT. *Science in China (Series E)*, 27(2):179–186, 1997.

[HDvMvZ04] Marijn Heule, Mark Dufour, Hans van Maaren, and Joris van Zwieten. March_eq: Implementing efficiency and additional reasoning into a lookahead SAT-solver. *Journal on Satisfiability, Boolean Modeling and Computation*, pages 25–30, 2004.

[HJ90] Pierre Hansen and Brigitte Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[HL06a] Federico Heras and Javier Larrosa. Intelligent variable orderings and re-orderings in DAC-based solvers for WCSP. *Journal of Heuristics*, 12(4–5):287–306, 2006.

[HL06b] Federico Heras and Javier Larrosa. New inference rules for efficient Max-SAT solving. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, Boston/MA, USA, 2006. AAAI Press.

[Hoo99] Holger Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI 1999)*, pages 661–666, Orlando/FL, USA, 1999. AAAI Press.

[HS04] Holger Hoos and Thomas Stützle. *Stochastic Local Search. Foundations and Applications*. Morgan Kaufmann, 2004.

[HTH02] Frank Hutter, Dave Tompkins, and Holger Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *LNCS*, pages 233–248. Springer, 2002.

[Hua07] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2318–2323, 2007.

[HV95] John N. Hooker and V Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[JMB97] Steve Joy, John E. Mitchell, and Brian Borchers. A branch and cut algorithm for MAX-SAT and Weighted MAX-SAT. In *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 519–536. American Mathematical Society, 1997.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[Kau06] Henry Kautz. Deconstructing planning as satisfiability. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, Boston/MA, USA, 2006. AAAI Press.

[KHR+02] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, , and Bart Selman. Dynamic restart policies. In *Procedings of the 18th National Conference on Artificial Intelligence (AAAI 2002)*, 2002.

[KRA+01] Henry Kautz, Yongshao Ruan, Dimitri Achlioptas, Carla Gomes, Bart Selman, and Mark Stickel. Balance and filtering in structured satisfiable problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 351–358, Seattle/WA, USA, 2001. Morgan Kaufmann.

[Kre88] Mark Krentel. The complexity of optimization problems. *Journal of Computer and Systems Sciences*, 36:490–509, 1988.

[Kri85] Balakrishnan Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22(3):253–275, 1985.

[KS96] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1996)*, pages 1194–1201, Portland/OR, USA, 1996. AAAI Press.

[KSTW05] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the 20th National Conference in Artificial Intelligence (AAAI 2005)*, pages 1368–1373, Pittsburgh/PA, USA, 2005. AAAI Press.

[LA97a] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 1997)*, pages 366–371, Nagoya, Japan, 1997. Morgan Kaufmann.

[LA97b] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the 3rd International Conference on Principles of Constraint Programming (CP 1997)*, volume 1330 of *LNCS*, pages 341–355, Linz, Austria, 1997. Springer.

[LAS05] Mark Liffiton, Zaher Andraus, and Karem Sakallah. From Max-SAT to Min-UNSAT: Insights and applications. Technical Report CSE-TR-506-05, University of Michigan, 2005.

[LD60] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[LH05a] Javier Larrosa and Federico Heras. Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 193–198, Edinburgh, Scotland, 2005. Morgan Kaufmann.

[LH05b] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *LNCS*, pages 158–172, St. Andrews, Scotland, 2005. Springer.

[Li03] Chu Min Li. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130:251–276, 2003.

[LLZ02] Michael Lewin, Dror Livnat, and Uri Zwick. Improved rounding techniques for the MAX 2-SAT and MAX DI-CUT problems. In *Ninth Conference on Integer Programming and Combinatorial Optimization (IPCO 2002)*, volume 2337 of *LNCS*, pages 67–82, Cambridge/MA, USA, 2002. Springer.

[LM02] Javier Larrosa and Pedro Meseguer. Partition-based lower bound for Max-CSP. *Constraints*, 7(3–4):407–419, 2002.

[LMP05] Chu Min Li, Felip Manyà, and Jordi Planes. Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In P. van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005)*, volume 3709 of *LNCS*, pages 403–414, Sitges, Spain, 2005. Springer.

[LMP06] Chu Min Li, Felip Manyà, and Jordi Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, pages 86–91, Boston/MA, USA, 2006. AAAI Press.

[LMS99] Javier Larrosa, Pedro Meseguer, and Thomas Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1):149–163, 1999.

[LMS01] Inês Lynce and João Marques-Silva. Integrating simplification techniques in SAT algorithms. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Boston/MA, USA, 2001. Short paper session.

[LMS02] Inês Lynce and João Marques-Silva. Efficient data structures for backtrack search SAT solvers. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*, pages 308–315, Cincinnati, USA, 2002.

[LMS05] Inês Lynce and João Marques-Silva. Efficient data structures for backtrack search SAT solvers. *Annals of Mathematics and Artificial Intelligence*, 47(1):137–152, 2005.

[LMSK63] J. Little, G. Murty, W. Sweeney, and C. Karel. An algorithm for the travelling salesman problem. *Operations Research*, 11:972–989, 1963.

[LS03] Javier Larrosa and Thomas Schiex. In the quest of the best form of local consistency for weighted CSP. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 239–244, Acapulco, Mexico, 2003. Morgan Kaufmann.

[LS04] Javier Larrosa and Thomas Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1–2):1–26, 2004.

[LW66] Eugene Lawler and D. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[Lyn04] Inês Lynce. *Propositional Satisfiability: Techniques, Algorithms and Applications*. PhD thesis, Instituto Superior Técnico. Universidade Técnica de Lisboa, October 2004.

[Mal05] Amol Dattatraya Mali. On quantified weighted MAX-SAT. *Decision Support Systems*, 40:257–268, 2005.

[McG79] James J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Informartion Sciences*, 19:229–250, 1979.

[MFM04] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In H. Hoos and D. Mitchell, editors, *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, volume 3542 of *LNCS*, pages 360–375, Vancouver, Canada, 2004. Springer. (Selected papers).

[MG04] Monaldo Mastrolilli and Luca Maria Gambardella. MAX-2-SAT: How good is tabu search in the worst-case? In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004)*, pages 173–178. AAAI Press, 2004.

[MG05] Monaldo Mastrolilli and Luca Maria Gambardella. Maximum satisfiability: How good are tabu search and plateau moves in the worst-case? *European Journal of Operational Research*, 166:63–76, 2005.

[MIK96] Shuichi Miyazaki, Kazuo Iwama, and Yahiko Kambayashi. Database queries as combinatorial optimization problems. In *Proceedings of International Symposium on Cooperative Database Systems for Advanced Applications*, pages 477–483, 1996.

[Mit05] David Mitchell. A SAT solver primer. *European Association for Theoretical Computer Science (EATCS) Bulletin*, 85:112–133, 2005.

[MMZ+01] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535, Las Vegas/NV, USA, 2001.

[MRS06] Pedro Meseguer, Francesca Rossi, and Thomas Schiex. Soft constraints. In F. Rossi, P. Van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 9. Elsevier, 2006.

[MS99] João Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In P. Barahona and J.J. Alferes, editors, *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence (EPIA 1999)*, volume 1695 of *LNCS*, pages 62–74, Évora, Portugal, 1999. Springer.

[MSG97] Bertrand Mazure, Lakhdar Saïs, and Éric Grégoire. Tabu search for SAT. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997)*, pages 281–285, Providence/RI, USA, 1997. AAAI Press.

[MSG99] João Marques-Silva and Luís Guerra. Algorithms for satisfiability in combinational circuits based on backtrack search and recursive learning. In *Proceedings of the 12th Symposium on Integrated Circuits and Systems Design (SBCCI 1999)*, 1999.

[MSK97] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997)*, pages 321–326, Providence/RI, USA, 1997. AAAI Press.

[MSS96a] João Marques-Silva and Karem Sakallah. Conflict analysis in search algorithms for satisfiability. In *Proceedings of 8th International Conference on Tools with Artificial Intelligence (ICTAI 1996)*, pages 467–469, Toulouse, France, 1996.

[MSS96b] João Marques-Silva and Karem Sakallah. GRASP – a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD 1996)*, pages 220–227, San Jose/CA, USA, 1996. IEEE Computer Society.

[MSS99] João Marques-Silva and Karem Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[NE03] Niklas Sörensson Niklas Eén. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, pages 502–518, Santa Margherita Ligure – Portofino, Italy, 2003.

[NLBH+04] Eugene Nudelman, Kevin Leyton-Brown, Holger Hoos, Alex Devkar, and Yoav Shoham. Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In M. Wallace, editor, *Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 438–452, Toronto, Canada, 2004. Springer.

[NR00] Rolf Niedermeier and Peter Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36:63–88, 2000.

[PL06] Steven Prestwitch and Inês Lynce. Local search for unsatisfiability. In A. Biere and C. Gomes, editors, *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, volume 4121 of *LNCS*, pages 283–296, Seattle/WA, USA, 2006. Springer.

[PR02] Panos M. Pardalos and Mauricio G. C. Resende, editors. *Handbook of Applied Optimization*, chapter 3.3, pages 65–77. Oxford University Press, 2002. Chapter writen by John E. Mitchell.

[Pre93] Daniele Pretolani. Efficiency and stability of hypergraph SAT algorithms. In *Proceedings of the DIMACS Challenge II Workshop*, 1993.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[Rya04] Lawrence Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[SB04] Laurent Simon and Daniel Le Berre. The SAT 2004 competition. In H. Hoos and D. Mitchell, editors, *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, volume 3542 of *LNCS*, pages 321–344, Vancouver, Canada, 2004. Springer. (Selected papers).

[SBH05] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT 2002 competition. *Annals of Mathematics and Artificial Intelligence*, 43(1):307–342, 2005.

[SBO07] Barbara Smith, Stefano Bistarelli, and Barry O'Sullivan. Breaking soft conditional symmetry. In *Proceedings of the First International Symmetry Conference*, Edinburgh, Scotland, 2007.

[Sch89] Uwe Schöning. *Logic for Computer Scientists*, volume 8 of *Progress in Computer Science and Applied Logic*. Birkhäuser, 1989.

[SHR01] Thomas Stützle, Holger Hoos, and Andrea Roli. A review of the literature on local search algorithms for MAX-SAT. Technical Report AIDA-01-02, FG Intellektik, FB Informatik, TU Darmstadt, Germany, 2001.

[SHS03] Kevin Smyth, Holger Hoos, and Thomas Stützle. Iterated robust tabu search for MAX-SAT. In Y. Xiand and B. Chaib-draa, editors, *Proceedings for the 16th Canadian Conference on Artificial Intelligence (AI 2003)*, volume 2671 of *LNCS*, pages 129–144, Halifax, Nova Scotia, Canada., 2003. Springer.

[SK93] Bart Selman and Henry Kautz. Domain-independent extensions of GSAT: Solving large structured satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 1993)*, pages 290–295, Chambery, France, 1993. Morgan Kaufmann.

[SKC94] Bart Selman, Henry Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*, pages 337–343, Seattle/WA, USA, 1994. AAAI Press.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI 1992)*, pages 440–446, San Jose/CA, USA, 1992. AAAI Press.

[SW02] John Slaney and Toby Walsh. Phase transition behaviour: from decision to optimization. In *Proceedings of 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*, 2002.

[SZ04] Haiou Shen and Hantao Zhang. Study of lower bound functions for MAX-2-SAT. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 2004)*, pages 185–190, San Jose, California, 2004. AAAI Press / MIT Press.

[SZ05] Haiou Shen and Hantao Zhang. Improving exact algorithms for MAX-2-SAT. *Annals of Mathematics and Artificial Intelligence*, 44:419–436, 2005.

[TH04] Dave Tompkins and Holger Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 306–320, 2004.

[TS02] Orestis Telelis and Panagiotis Stamatopoulos. Heuristic backbone sampling for maximum satisfiability. In *Proceedings of the 2nd Hellenic Conference on Artificial Intelligence (SETN 2002)*, pages 129–139, Thessaloniki, Greece, 2002.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[VB03] Miroslav N. Velev and Randal E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[Vel89] André Vellino. *The Complexity of Automated Reasoning*. PhD thesis, University of Toronto, 1989.

[vMvN05] Hans van Maaren and Linda van Norden. Sums of squares, satisfiability and maximum satisfiability. In F. Bachus and T. Walsh, editors, *International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *LNCS*, pages 294–308, St. Andrews, Scotland, 2005. Springer.

[WF96] Richard J. Wallace and Eugene Freuder. Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. In D. Johnson and M. Trick, editors, *Cliques, Coloring and Satisfiability*, volume 26, pages 587–615. American Mathematical Society, 1996.

[WvM98] Joost P. Warners and Hans van Maaren. A two-phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters*, 23:81–88, 1998.

[XZ04] Zhao Xing and Weixiong Zhang. Efficient strategies for (weighted) maximum satisfiability. In *Proceedings of International conference on principles and practice of constraint programming (CP 2004)*, volume 3258 of *LNCS*, pages 690–705, Toronto, Canada, 2004. Springer.

[XZ05] Zhao Xing and Weixiong Zhang. An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence*, 164(2):47–80, 2005.

[Yan94] Mihalis Yannakakis. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17:475–502, 1994.

[Zha97] Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the Conference on Automated Deduction (CADE 1997)*, volume 1249 of *LNCS*, pages 272–275, Townsville, North Queensland, Australia, 1997. Springer.

[Zha03] Lintao Zhang. *Searching for truth: techniques for satisfiability of boolean formulas*. PhD thesis, Department of Electrical Engineering. Princeton University, June 2003.

[ZM88] Ramin Zabih and David A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 1988)*, pages 155–160, Saint Paul/MN, USA, 1988.

[ZM02] Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In *Proceedings of the 18th International Conference on Automated Deduction (CADE 2002)*, volume 2392 of *LNCS*, pages 295–313, Copenhagen, Denmark, 2002. Springer.

[ZRL03] Weixiong Zhang, Ananda Rangan, and Moshe Looks. Backbone guided local search for maximum satisfiability. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1179–1186, Acapulto (Mexico), August 2003. Morgan Kaufmann.

[ZS96] Hantao Zhang and Mark E. Stickel. An efficient algorithm for unit propagation. In M. Golumbic, editor, *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH 1996)*, Fort Lauderdale/FL, USA, 1996.

[ZSM03a] Hantao Zhang, Haiou Shen, and Felip Manyà. Exact algorithms for MAX-SAT. *Electronic Notes in Theoretical Computer Science*, 86(1):190–203, 2003.

[ZSM03b] Hantao Zhang, Haiou Shen, and Felip Manyà. Exact algorithms for MAX-SAT. In *Proceedings of the 4th International Workshop on First order Theorem Proving (FTP 2003)*, pages 133–146, Valencia, Spain, June 2003.

Monografies de l'Institut d'Investigació en Intel·ligència Artificial

Num. 1	J. Puyol, *MILORD II: A Language for Knowledge–Based Systems*
Num. 2	J. Levy, *The Calculus of Refinements, a Formal Specification Model Based on Inclusions*
Num. 3	Ll. Vila, *On Temporal Representation and Reasoning in Knowledge–Based Systems*
Num. 4	M. Domingo, *An Expert System Architecture for Identification in Biology*
Num. 5	E. Armengol, *A Framework for Integrating Learning and Problem Solving*
Num. 6	J. Ll. Arcos, *The Noos Representation Language*
Num. 7	J. Larrosa, *Algorithms and Heuristics for Total and Partial Constraint Satisfaction*
Num. 8	P. Noriega, *Agent Mediated Auctions: The Fishmarket Metaphor*
Num. 9	F. Manyà, *Proof Procedures for Multiple-Valued Propositional Logics*
Num. 10	W. M. Schorlemmer, *On Specifying and Reasoning with Special Relations*
Num. 11	M. López-Sánchez, *Approaches to Map Generation by means of Collaborative Autonomous Robots*
Num. 12	D. Robertson, *Pragmatics in the Synthesis of Logic Programs*
Num. 13	P. Faratin, *Automated Service Negotiation between Autonomous Computational Agents*
Num. 14	J. A. Rodríguez, *On the Design and Construction of Agent-mediated Electronic Institutions*
Num. 15	T. Alsinet, *Logic Programming with Fuzzy Unification and Imprecise Constants: Possibilistic Semantics and Automated Deduction*
Num. 16	A. Zapico, *On Axiomatic Foundations for Qualitative Decision Theory A Posibilistic Approach*
Num. 17	A. Valls, *ClusDM: A multiple criteria decision method for heterogeneous data sets*
Num. 18	D. Busquets, *A Multiagent Approach to Qualitative Navigation in Robotics*
Num. 19	M. Esteva, *Electronic Institutions: from specification to development*
Num. 20	J. Sabater, *Trust and reputation for agent societies*

Num. 21 J. Cerquides, *Improving Algorithms for Learning Bayesian Network Classifiers*

Num. 22 M. Villaret, *On Some Variants of Second-Order Unification*

Num. 23 M. Gómez, *Open, Reusable and Configurable Multi-Agent Systems: A Knowledge Modelling Approach*

Num. 24 S. Ramchurn, *Multi-Agent Negotiation Using Trust and Persuasion*

Num. 25 S. Ontañón, *Ensemble Case-Based Learning for Multi-Agent Systems*

Num. 26 M. Sánchez, *Contributions to Search and Inference Algorithms for CSP and Weighted CSP*

Num. 27 C. Noguera, *Algebraic Study of Axiomatic Extensions of Triangular Norm Based Fuzzy Logics*

Num. 28 E. Marchioni, *Functional Definability Issues in Logics Based on Triangular Norms*

Num. 29 M. Grachten, *Expressivity-Aware Tempo Transformations of Music Performances Using Case Based Reasoning*

Num. 30 I. Brito, *Distributed Constraint Satisfaction*

Num. 31 E. Altamirano, *On Non-clausal Horn-like Satisfiability Problems*

Num. 32 A. Giovannucci, *Computationally Manageable Combinatorial Auctions for Supply Chain Automation*

Num. 33 R. Ros, *Action Selection in Cooperative Robot Soccer using Case-Based Reasoning*

Num. 34 A. García Cerdaña, *On some Implication-free Fragments of Substructural and Fuzzy Logics*

Num. 35 A. García Camino, *Normative Regulation of Open Multi-agent Systems*

Num. 36 A. Robles, *Enabling Intelligent Organizations: An Electronic Institutions Approach for Building Agent Oriented Information Systems*

Num. 37 I. Drummond, *Imprecise Classification Based on Fuzzy Logic and Possibility Theory*

Num. 38 J. Planes, *Design and Implementation of Exact MAX-SAT Solvers*