UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Parallel Search for Maximum Satisfiability

Rúben Carlos Gonçalves Martins

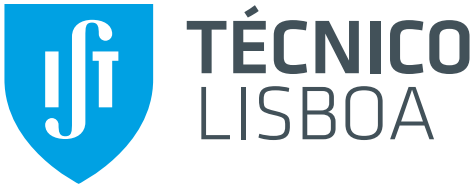
Supervisor: Doctor Maria Inês Camarate de Campos Lynce de Faria
Co-Supervisor: Doctor Vasco Miguel Gomes Nunes Manquinho

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering
Pass With Distinction

Jury:

Chairperson: Chairman of the IST Scientific Board
Members of the Committee:
Doctor Salvador Luís de Bethencourt Pinto de Abreu
Doctor Maria Inês Camarate de Campos Lynce de Faria
Doctor Vasco Miguel Gomes Nunes Manquinho
Doctor João Manuel Pinheiro Cachopo
Doctor Youssef Hamadi

2013



UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Parallel Search for Maximum Satisfiability

Rúben Carlos Gonçalves Martins

Supervisor: Doctor Maria Inês Camarate de Campos Lynce de Faria
Co-Supervisor: Doctor Vasco Miguel Gomes Nunes Manquinho

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering
Pass With Distinction

Jury:

Chairperson: Chairman of the IST Scientific Board
Members of the Committee:

Doctor Salvador Luís de Bethencourt Pinto de Abreu
Professor Associado (com Agregação)
Escola de Ciências e Tecnologia, Universidade de Évora

Doctor Maria Inês Camarate de Campos Lynce de Faria
Professora Auxiliar
Instituto Superior Técnico, Universidade de Lisboa

Doctor Vasco Miguel Gomes Nunes Manquinho
Professor Auxiliar
Instituto Superior Técnico, Universidade de Lisboa

Doctor João Manuel Pinheiro Cachopo
Professor Auxiliar
Instituto Superior Técnico, Universidade de Lisboa

Doctor Youssef Hamadi
Senior Researcher
Microsoft Research, Cambridge, United Kingdom

Funding Institutions:

Fundação para a Ciência e Tecnologia
INESC-ID

2013

In memory of my father,

You taught me never to give up,

To always fight for what I believed in,

You will never be forgotten.

Resumo

A predominância de processadores com múltiplos núcleos tem aumentado o interesse em desenvolver ferramentas paralelas para Satisfação Booleana (SAT). Como consequência, tem vindo a ser desenvolvido um número cada vez maior de ferramentas paralelas As abordagens paralelas são conhecidas por aumentar a performance das ferramentas sequenciais. No entanto, existe um número muito reduzido de ferramentas paralelas para optimização Booleana.

Esta dissertação propõe novos algoritmos paralelos para resolver o problema de Satisfação Máxima (MaxSAT) e apresenta a primeira ferramenta paralela para MaxSAT, denominada PWBO. A ferramenta PWBO pode usar duas estratégias distintas para fazer uma procura em paralelo. A primeira estratégia usa uma abordagem portfólio. Esta abordagem realiza a procura nos valores dos limites inferior e superior da solução óptima. Para diversificar a procura, cada processo usa uma codificação diferente para restrições de cardinalidade. A segunda estratégia divide o espaço de procura considerando diferentes valores do limite superior da solução óptima.

Tal como outras ferramentas paralelas, PWBO tem um comportamento não determinístico, ou seja, várias execuções da mesma ferramenta podem resultar em soluções diferentes. Esta é uma desvantagem clara para aplicações que precisam de resolver a mesma instância de um problema mais que uma vez. Por este motivo, apresentamos a primeira ferramenta paralela determinística para MaxSAT. Finalmente, também propomos técnicas de partição para melhorar o desempenho de algoritmos sequenciais para MaxSAT.

Palavras-chave: Satisfação Booleana (SAT), Optimização Booleana, Satisfação Máxima (MaxSAT), Restrições de Cardinalidade, Procura Paralela, Procura Paralela Determinística, Partilha de Cláusulas, Algoritmos baseados em No Satisfação, Partições, Comunidades em Grafos

Abstract

The predominance of multicore processors has increased the interest in developing parallel Boolean Satisfiability (SAT) solvers. As a result, more parallel SAT solvers are emerging. Even though parallel approaches are known to boost performance, parallel solvers developed for Boolean optimization are scarce.

This dissertation proposes parallel search algorithms for Maximum Satisfiability (MaxSAT) and introduces PWBO, the first parallel solver for MaxSAT. PWBO can use two different strategies for parallel search. The first strategy performs a portfolio approach by searching on the lower and upper bound values of the optimal solution using different encodings of cardinality constraints for each thread. The second strategy splits the search space considering different upper bound values of the optimal solution for each thread.

As others parallel solvers, PWBO suffers from non-deterministic behavior, i.e. several runs of the same solver can lead to different solutions. This is a clear downside for applications that require solving the same problem instance more than once. Therefore, we also present the first deterministic parallel MaxSAT solver that ensures reproducibility of results. Finally, we also propose partitioning techniques to improve sequential MaxSAT algorithms.

Keywords: Boolean Satisfiability (SAT), Boolean Optimization, Maximum Satisfiability (MaxSAT), Cardinality Constraints, Parallel Search, Deterministic Parallel Search, Clause Sharing, Unsatisfiability-based Algorithms, Partitioning, Graph Communities

Acknowledgments

"Success is almost totally dependent upon drive and persistence. The extra energy required to make another effort or try another approach is the secret of winning."

Denis Waitley

Finally it is done! The thirst for knowledge kept me going for the last four years. In these years, I have grown as a person and as a researcher. However, this process would not be possible without the help of several people.

First of all, I would like to thank my advisors, Professor Inês Lynce and Professor Vasco Manquinho. I know Professor Inês Lynce since my MSc and she has always inspired me to do better. She is a great researcher and a role model. Professor Vasco Manquinho introduced me to the field of Boolean optimization. He has been a dedicated advisor and has always supported me. He taught me how to organize my research and how to question everything. I have learned a lot with him and I will always be in his debt.

I would like to thank my mother Prazeres for all her sacrifices that allowed me to follow my dreams and achieve my goals. I would also like to thank my sister Sara for her daily support. Without them I would not be able to finish this thesis.

I would also like to thank all my colleagues, in particularly Mikoláš Janota for the many insightful discussions. Not only he is a great researcher but also a good friend that helped me in my good and bad days.

Finally, I would like to thank all the institutions that supported this research work. In particular, the Fundação para a Ciência e Tecnologia through the research projects iExplain (PTDC/EIA-CCO/102077/2008) and BSOLO (PTDC/EIA/76572/2006), and INESC-ID multiannual funding through the PIDDAC program funds.

Contents

List of Tables

List of Figures

List of Algorithms

1

Introduction

Suppose we have to plan a birthday dinner and we want to invite three out of five friends. However, not all friends get along. Therefore, if we want to have a friendly dinner we must take into consideration the relationships between them. Suppose we have five friends: Angela, Bob, Charles, David and Eve. Moreover, consider we have the following constraints: (i) Angela does not talk with Bob, (ii) Charles is upset with David, and (iii) Eve does not like Angela.

How can we solve this puzzle? This is a simple logic problem that can be written as a propositional formula. Even though, propositional logic has limited expressiveness, it is powerful enough to encode many real-world problems.

A propositional formula is typically formulated as a conjunction of clauses where each clause is a disjunction of literals. A variable x can be assigned truth value 0 or 1, and a literal $\bar{x}$ denotes the complement of x. Consider that the variables a (Angela), b (Bob), c (Charles), d (David) and e (Eve) represent the friends.

The following clauses encode the constraints between the friends:

- If we invite Angela, then we do not invite Bob: $(\bar{a} \vee \bar{b})$,
- If we invite Charles, then we do not invite David: $(\bar{c} \vee \bar{d})$,
- If we invite Eve, then we do not invite Angela: $(\bar{e} \vee \bar{a})$.

Additionally, we also want to encode that we want to invite three friends. Note that, if we do not invite two of the friends then it implies that the remaining three must be invited. For

example, if we do not invite Angela and Bob, then we must invite Charlie, David and Eve. The following set of clauses encode this fact:

- $(a \vee b \vee c)$
- $(a \vee b \vee d)$
- $(a \vee b \vee e)$
- $(a \vee c \vee d)$
- $(a \vee c \vee e)$
- $(a \vee d \vee e)$
- $(b \vee c \vee d)$
- $(b \vee c \vee e)$
- $(b \vee d \vee e)$
- $(c \vee d \vee e)$

Can we find an assignment to the problems variables such that all clauses are satisfied? This problem is a decision problem known as Boolean Satisfiability (SAT).

What is SAT?

Given a propositional formula, the Boolean Satisfiability (SAT) problem consists of deciding whether there exists an assignment to the problem variables such that all the clauses are satisfied or prove that no assignment which satisfies all clauses exists.

One possible assignment to the problem variables that satisfies the propositional formula that encodes the birthday problem presented above is $a = 0$, $b = 1$, $c = 1$, $d = 0$ and $e = 1$. This assignment means that we invite Bob, Charles and Eve to the birthday dinner.

SAT is known for its theoretical importance. SAT was the first problem to be proved to be NP-Complete [Coo71]. Moreover, SAT has many applications in real-world problems, including planning [KS92], hardware and software verification [BCCZ99], general theorem proving [MB08], and computational biology [CBH$^+$07]. The widespread use of SAT is the result of SAT solvers being so effective in practice. Many industrial problems with hundreds of thousands of variables and millions of clauses can now be solved within a few minutes.

The birthday problem presented previously is satisfiable. However, for some SAT instances there are no assignments that satisfy all clauses. For example, if we had the same birthday problem but wanted to invite four friends instead of three, then we will not be able to find an assignment to the problem variables that satisfy all clauses since no such assignment exists.

For some real-world applications, we may require an assignment even if it does not satisfy all clauses. For example, we may be interested in an assignment that maximizes the number of satisfied clauses. For these applications, we may consider using Maximum Satisfiability (MaxSAT).

What is MaxSAT?

Maximum Satisfiability (MaxSAT) is an optimization version of SAT where the goal is to find an assignment to the problem variables such that the number of satisfied (unsatisfied) clauses is maximized (minimized).

Consider the formula composed with the following clauses:

- $(\bar{a} \vee \bar{b})$
- $(\bar{c} \vee \bar{d})$
- $(\bar{e} \vee \bar{a})$
- $(a \vee b)$
- $(a \vee c)$
- $(a \vee d)$
- $(a \vee e)$
- $(b \vee c)$
- $(b \vee d)$
- $(b \vee e)$
- $(c \vee d)$
- $(c \vee e)$
- $(d \vee e)$

The above formula encodes the birthday problem described previously where we want to invite four friends. This formula is unsatisfiable, i.e there are no assignments that satisfy all clauses. However, it is possible to satisfy all clauses but one. For example, the assignment $a = 0$, $b = 1$, $c = 1$, $d = 1$, $e = 1$ satisfies all clauses except $(\bar{c} \vee \bar{d})$.

MaxSAT has several variants. For example, one may consider a propositional formula with two sets of clauses: hard clauses and soft clauses. Hard clauses are clauses that must be satisfied, whereas soft clauses are clauses that may or not be satisfied. The problem of satisfying all hard clauses and maximizing the number of satisfied soft clauses is called the partial MaxSAT problem.

Consider again the unsatisfiable formula presented above. Suppose that the clause $(\bar{c} \vee \bar{d})$ is hard and the remaining clauses are soft. Therefore, we want to find an assignment that satisfies the clause $(\bar{c} \vee \bar{d})$ and maximizes the number of satisfied soft clauses. The assignment $a = 0$, $b = 1$, $c = 1$, $d = 0$, $e = 1$ satisfies the hard clause $(\bar{c} \vee \bar{d})$ and satisfies all soft clauses except $(a \vee d)$.

Finally, MaxSAT can be further generalized by associating a weight to each clause. Therefore, the weighted MaxSAT problem consists in maximizing the sum of the weights of the satisfied clauses.

Why is MaxSAT important?

Nowadays, MaxSAT solvers are powerful tools that can be used in many important application domains, such as software package upgrades [ABL+10b], error localization in C code [JM11], debugging of hardware designs [CSMSV10], haplotyping with pedigrees [GLMSO10], and course timetabling [AN12].

Table 1.1: Example of a software package upgrade problem

Package	Dependencies	Conflicts
p_1	$\{p_2 \vee p_3\}$	$\{p_4\}$
p_2	$\{p_3\}$	$\{\}$
p_3	$\{p_2\}$	$\{p_4\}$
p_4	$\{p_2 \wedge p_3\}$	$\{\}$

For example, consider the software package upgradeability problem [MBC+06] where we have a set of packages we want to install. Each package p_i has a set of dependencies and a set of conflicts. The dependencies denote packages which p_i depends on. Therefore, those packages must be installed for p_i to be installed. On the other hand, conflicts denote packages which cannot be installed for p_i to be installed.

Table 1.1 shows an example of a software package upgradeability problem instance, where we have four packages $\{p_1, p_2, p_3, p_4\}$. Each package has a set of dependencies and a set of conflicts. If we consider the problem presented in Table 1.1 we cannot install all packages. Note that package p_4 requires package p_3 to be installed, but at the same time package p_3 has a conflict with package p_4. However, we still want to install some of the packages into our system. Therefore, consider we want to maximize the number of packages installed in our system.

This problem can be encoded as a partial MaxSAT problem, where we have a set of hard clauses and a set of soft clauses. Consider the following hard clauses:

$$(\bar{p}_1 \vee p_2 \vee p_3) \wedge (\bar{p}_1 \vee \bar{p}_4) \wedge (\bar{p}_2 \vee p_3) \wedge (\bar{p}_3 \vee p_2) \wedge (\bar{p}_3 \vee \bar{p}_4) \wedge (\bar{p}_4 \vee p_2) \wedge (\bar{p}_4 \vee p_3) \tag{1.1}$$

These clauses correspond to the dependencies and conflicts between the different packages. For example, $(\bar{p}_1 \vee p_2 \vee p_3)$ corresponds to the dependencies of package p_1, i.e. if p_1 is installed, then either p_2 or p_3 must also be installed. On the other hand, $(\bar{p}_1 \vee \bar{p}_4)$ corresponds to the conflicts of package p_1, i.e. if p_1 is installed then p_4 cannot be installed.

Since we want to maximize the number of installed packages, we include that information in the formula by using the following soft clauses:

$$(p_1) \wedge (p_2) \wedge (p_3) \wedge (p_4) \tag{1.2}$$

Therefore, in the resulting partial MaxSAT problem we have to satisfy all clauses in Equation (1.1), while maximizing the number of satisfied clauses in Equation (1.2). The assignment

Table 1.2: Number of parallel SAT solvers that participated in the annual SAT competitions

	SAT Competitions				
	2008	2009	2010	2011	2012
#Parallel SAT solvers	3	3	6	12	19

$p_1 = 1$, $p_2 = 1$, $p_3 = 1$ and $p_4 = 0$ satisfies all clauses in Equation (1.1), while satisfying 3 out of 4 clauses in Equation (1.2).

MaxSAT can be used to effectively solve the software package upgradeability problem. For example, the widely used Eclipse platform[1] uses MaxSAT for managing the plugins dependencies [BP10].

Several real-world problems can be encoded into MaxSAT and solved by MaxSAT solvers. Therefore, improving MaxSAT algorithms will result in more effective optimization solvers with a strong expected impact on several application areas.

Why parallel MaxSAT?

Nowadays, extra computational power is no longer coming from higher processor frequencies. At the same time, multicore architectures are becoming predominant. Exploiting this new architecture is essential for the evolution of SAT and MaxSAT solvers.

In the last years, there has been an increase in the research of parallel SAT solving, particularly for multicore architectures. Table 1.2 shows the number of parallel SAT solvers that participated in the SAT competitions from 2008 to 2012[2–6]. As we can see in Table 1.2, the number of parallel SAT solvers has been increasing during the last years. For example, 3 parallel SAT solvers participated in the competition of 2008, whereas 19 parallel SAT solvers participated in the competition of 2012.

Following the recent success of parallel SAT solvers, it comes as a natural step to develop parallel algorithms for MaxSAT. Although this reasoning comes as natural, there are no parallel implementations for solving MaxSAT. Even on related Boolean optimization fields, there are only a few parallel implementations. In the context of Pseudo-Boolean Optimization (PBO), SAT4J PB RES//CP[7], implements a resolution based algorithm that competes with a cutting plane based

[1] http://www.eclipse.org/
[2] http://baldur.iti.uka.de/sat-race-2008/
[3] http://www.satcompetition.org/2009/
[4] http://baldur.iti.uka.de/sat-race-2010/
[5] http://www.satcompetition.org/2011/
[6] http://baldur.iti.kit.edu/SAT-Challenge-2012/
[7] http://www.satcompetition.org/PoS/presentations-pos/leberre.pdf

algorithm. In the context of Integer Linear Programming (ILP), the commercial solver CPLEX[8] is known to have the option of performing parallel search but no detailed description is available.

What is the focus of this thesis?

In this work we have taken the first steps towards developing parallel MaxSAT algorithms for multicore architectures. By using a small number of cores we are able to improve the performance of sequential MaxSAT algorithms. Improving MaxSAT algorithms is expected to have a strong impact on the scientific community, as well as on several application areas.

This dissertation focus on the development of the first parallel algorithms for MaxSAT. Our goal is to improve MaxSAT algorithms, and therefore we do not focus on improving data structures or locking mechanisms. Each thread of the parallel solvers presented in this dissertation has a private copy of the formula and a simple locking system is used to share information between threads. Note that this locking system does not impose an overhead on our system.

In what follows we describe the main contributions of this dissertation and explain how it is organized.

1.1 Contributions

This dissertation contains several contributions on MaxSAT solving. Moreover, the chapter on Boolean Satisfiability also involved research work, that resulted in two publications: a paper with improvements to search space splitting that was published in the proceedings of the *International Conference on Tools with Artificial Intelligence* [MML10b], and an overview on parallel SAT solving that was published in *Constraints* [MML12a]. This work has been fundamental to understand the related work in parallel SAT solving. Next, we describe each of the main contributions on MaxSAT solving.

Dynamic encoding heuristic for cardinality constraints

Cardinality constraints are usually used by MaxSAT algorithms. This dissertation evaluates a large number of cardinality encodings and proposes a dynamic encoding heuristic that selects the most adequate encoding for a given instance. The evaluation of the different cardinality encodings was published in the proceedings of the *International Conference on Tools with Artificial*

[8] `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

Intelligence [MML11a], whereas the dynamic encoding heuristic was published in *AI Communications* [MML12e].

Parallel approaches for MaxSAT solving

This dissertation proposes parallel approaches for solving MaxSAT and introduces PWBO, the first parallel solver for MaxSAT. PWBO is targeted to multicore architectures and can use different parallel strategies. One strategy is based on a portfolio approach by searching on the lower and upper bound values of the optimal solution using different encodings of cardinality constraints for each thread. Another strategy is to split the search space considering different upper bound values of the optimal solution for each thread. A preliminary version of this work was presented at the *RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion* [MML11b]. Further improved versions were published in the proceedings of the *International Conference on Tools with Artificial Intelligence* [MML11a], and in *AI Communications* [MML12e].

Deterministic approaches for parallel MaxSAT solving

PWBO is a non-deterministic parallel MaxSAT solver. Even though PWBO is able to improve the performance of sequential MaxSAT solvers, it cannot be used in applications domains that require reproducible results. Therefore, we also present a deterministic version of PWBO, which ensures reproducibility of results. A preliminary version of this work was presented at the *RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion* [MML12b], whereas a further improved version is under review in *AI Communications* [MML13b].

Clause sharing heuristics for parallel MaxSAT

In parallel MaxSAT Solving, sharing learned clauses is expected to further prune the search space and boost the performance of a parallel solver. However, not all learned clauses should be shared since it would lead to an exponential blow up in memory and to sharing many irrelevant clauses. The problem of determining if a shared clause will be useful in the future remains challenging, and in practice heuristics are used to select which learned clauses should be shared. We propose a new heuristic based on the notion of freezing. This heuristic delays incorporating the shared learned clauses that were imported from other threads. These clauses are frozen until they are considered

relevant in the context of the current search. A preliminary version of this work was published in the proceedings of the *Learning and Intelligent Optimization Conference* [MML12c], whereas a further improved version is under review in *AI Communications* [MML13b].

Improving Sequential MaxSAT

Partitioning formulas is motivated by the expectation to identify easy to solve subformulas. We propose to apply partitioning to MaxSAT. The use of partitions can be naturally combined with unsatisfiability-based algorithms for MaxSAT that are built upon successive calls to a SAT solver, where each call identifies an unsatisfiable subformula. One of the drawbacks of these algorithms results from the SAT solver returning large unsatisfiable subformulas. However, when using partitions, the solver is more likely to identify smaller unsatisfiable subformulas. Different partitioning methods are proposed in this dissertation, namely, weight-based, graph-based and community-based. Partitioning has shown to significantly improve the performance of unsatisfiability-based algorithms for weighted partial MaxSAT and partial MaxSAT. A first version of this work for weighted partial MaxSAT was published in the proceedings of the *European Conference on Artificial Intelligence* [MML12d], whereas an enhanced version for partial MaxSAT was published in the proceedings of the *International Conference on Theory and Applications of Satisfiability Testing* [MML13a].

1.2 Organization

This dissertation has a total of eight chapters and it is organized as follows.

Chapter 1 starts with a small introduction to Boolean Satisfiability (SAT) and Maximum Satisfiability (MaxSAT). Additionally, we also describe the main contributions of our work and how this dissertation is organized. Chapters 2 and 3 formally describe the SAT and MaxSAT problems. Chapters 4, 5, 6 and 7 present the main contributions of this dissertation. Finally, chapter 8 concludes this dissertation and suggests future work.

Chapter 2 formally defines SAT and overviews sequential and parallel algorithms for SAT solving. For sequential SAT solving, we describe Conflict-Driven Clause learning (CDCL) SAT solvers. For parallel SAT solving, we describe the two most common approaches, portfolios and search space splitting.

Chapter 3 formally defines MaxSAT and overviews sequential MaxSAT algorithms. This chapter describes linear search algorithms and unsatisfiability-based algorithms for MaxSAT.

Moreover, this chapter also describes and evaluates a large number of encodings for cardinality constraints. Taking advantage of the diversity of encodings for cardinality constraints, we propose a dynamic encoding heuristic that selects the most adequate encoding for a given instance.

Chapter 4 introduces PWBO, the first parallel solver for MaxSAT. Different versions of PWBO are proposed. The first version uses two threads, one thread searching on the lower bound value of the optimal solution, and another thread searching on the upper bound value of the optimal solution. For a larger number of threads, two versions are proposed, namely, portfolio and splitting. The portfolio approach uses several threads to simultaneously search on the lower and upper bound values of the optimal solution. These threads differ between themselves in the encoding used for cardinality constraints. The splitting approach uses the additional threads to search on different values of the upper bound. To further increase the performance of PWBO, learned clauses are shared among the different threads. Experimental results show the effectiveness of our parallel MaxSAT solvers and show that sharing learned clauses further reduces the running times of PWBO.

Chapter 5 presents a deterministic version of PWBO. This solver can be used in applications that require reproducibility of results. Different approaches for thread synchronization are studied, namely, standard, period and dynamic synchronization. Dynamically changing the number of conflicts that each thread requires to reach a synchronization point is shown to improve the performance of the deterministic solver. Experimental results show that the performance of the deterministic solver is comparable to the corresponding non-deterministic solver.

Chapter 6 describes different sharing heuristic procedures that were already proposed for parallel SAT solving and integrate them into the deterministic parallel MaxSAT solver that was proposed in the previous chapter. Moreover, a new heuristic based on the notion of freezing is proposed in this chapter. This heuristic delays incorporating the shared learned clauses that were imported from other threads. These clauses are frozen until they are considered relevant in the context of the current search. By using a deterministic solver one can independently evaluate the gains coming from the use of different heuristics rather than the non-determinism of the solver. Experimental results show that sharing learned clauses improves the overall performance of parallel MaxSAT solvers.

Chapter 7 presents a new technique based on partitioning that can be used to enhance the performance of sequential unsatisfiability-based algorithms. Different partitioning methods are proposed in this chapter, namely, weight-based, graph-based and community-based. For MaxSAT instances with weights, weight-based partitioning has shown to be the most effective method of

partitioning. On the other hand, for MaxSAT instances without weights, graph-based partitioning and community-based partitioning has shown to be particularly effective. Experimental results show that partitioning significantly improves the performance of unsatisfiability-based algorithms.

Chapter 8 concludes the dissertation and suggests future research directions.

2

Boolean Satisfiability

In recent years, Boolean satisfiability (SAT) solvers have been successfully used in many practical applications, including planning [KS92], hardware and software verification [BCCZ99], general theorem proving [MB08], and computational biology [CBH+07].

The widespread use of SAT is the result of SAT solvers being very effective in practice. Even though real world problems tend to be large in size, SAT solvers are now able to solve many problems with hundreds of thousands of variables and millions of clauses in just a few minutes. However, there are still many problems that remain challenging for modern SAT solvers. For example, many industrial instances from the last competition remain unsolved. Even though SAT solvers are still improving, the gains given by small algorithmic adjustments are scarce.

The demand for more computational power led to the creation of new computer architectures composed by multiple machines connected by a network, such as clusters and grids. In the last decade, parallel SAT solving has been the target of research since using these new architectures allows solving more problem instances [BS96, ZBH96, BSK03]. Lately, with the predominance of multicore processors, the interest in parallel SAT solving has increased and more parallel SAT solvers are emerging [HJS09b, CS08, LSB07].

In this chapter we start by providing some definitions used throughout this dissertation. Afterwards, we describe the architecture of a modern SAT solver. Finally, we briefly describe the main approaches for parallel SAT solving.

2.1 Definitions

Boolean formulas are commonly represented in Conjunctive Normal Form (CNF). A CNF formula is represented using n Boolean variables $x_1, x_2, \ldots, x_n$, which can be assigned truth values *0* (false) or *1* (true). A literal l is either a variable x_i (i.e., a positive literal) or its complement $\bar{x}_i$ (i.e., a negative literal). A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses. For simplicity, in the remainder of the dissertation we will consider a formula as being a set of clauses.

Example 2.1. *Consider the following CNF formula:* $\varphi = \{(x_1 \vee x_2 \vee \bar{x}_3), (x_2 \vee x_3), (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)\}$. *This formula has 3 variables and 3 clauses.* $(x_2 \vee x_3)$ *has only positive literals, whereas* $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ *has only negative literals.*

A literal is *satisfied* if its truth value is 1 and *unsatisfied* if its truth value is 0. A literal with no truth value is said to be *unassigned*. A clause is said to be *satisfied* if at least one of its literals is satisfied, and it is said to be *unsatisfied* if all of its literals are unsatisfied. A clause with no literals, called an *empty* clause, is *unsatisfiable*. A clause is *unit* if all literals but one are unsatisfied, and the remaining literal is unassigned. If a clause is neither satisfied, unsatisfied or unit, it is said to be *unresolved*.

Definition 2.1 (Assignment)**.** *Given a Boolean formula* φ*, an assignment is a mapping* $\nu : X \rightarrow \{0, 1\}$ *defined on the set* X *of problem variables of* φ*.*

A formula is *satisfied* if all its clauses are satisfied by a given assignment to the problem variables. A formula is *unsatisfied* if at least one of its clauses is unsatisfied. A formula is *unsatisfiable* if there is no assignment that makes the formula satisfied.

For simplicity, in the remainder of the dissertation we will consider an assignment to a problem variable x_i, as either satisfying the positive literal l_i, or the negative literal $\bar{l}_i$.

Example 2.2. *Given the formula in example 2.1, the assignment* $\nu = \langle x_1, x_2, \bar{x}_3 \rangle$ *makes the formula satisfied and the assignment* $\nu' = \langle x_1, x_2, x_3 \rangle$ *makes the formula unsatisfied (*$(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ *is unsatisfied). The formula* $\varphi' = \{(x_1), (\bar{x}_1 \vee x_2), (\bar{x}_1 \vee \bar{x}_2)\}$ *is unsatisfiable since there is no assignment that satisfies all clauses.*

Definition 2.2 (Boolean Satisfiability)**.** *The Boolean Satisfiability (SAT) problem consists of deciding whether there exists an assignment to the variables such that the formula is satisfied. Such assignment is called a solution.*

SAT solvers can be divided into two categories: *complete* and *incomplete*. Given a CNF formula, complete SAT solvers can find a solution or prove that no solution exists. On the other hand, incomplete SAT solvers can prove either satisfiability [SKC96] or unsatisfiability [PL06b] but not both. Even though incomplete solvers can solve unsatisfiable instances they are not effective for these kind of instances. In practice, incomplete solvers are usually used to solve unstructured satisfiable problems. However, the majority of the industrial applications use a complete SAT solver since the instances have a defined structure and the solver is required to be able to prove satisfiability and unsatisfiability. For this reason, the remainder of this chapter focus on complete SAT solvers.

2.2 Sequential SAT Solving

The first well-known algorithm for solving SAT was based on resolution [Rob65] and was proposed by Davis and Putnam in 1960 [DP60]. The original algorithm suffers from the problem of memory explosion. Therefore, Davis, Logemann and Loveland proposed a modified version [DLL62] that used backtrack search to limit the memory required by the solver. This algorithm is often referred to as the DPLL algorithm and corresponds to a depth-first backtrack search where at each branching step a variable is chosen and assigned a truth value. These variables are denoted as decision variables. Each decision variable is assigned at a different decision level.

Definition 2.3 (Decision Level)**.** *A decision level of a variable denotes the depth of the search tree at which the variable is assigned a truth value.*

Following that, the logical consequences of each decision variable are evaluated. Each time an unsatisfied clause (also known as *conflict*) is identified, *backtracking* is executed. Backtracking corresponds to unassigning decision variables and their logical consequences until a decision variable is found where one of the truth values have not been tried. If both truth values have been considered for the decision variable that was assigned at decision level 1, and backtracking undoes this first decision variable, then the CNF formula can be declared unsatisfiable. This kind of backtracking is called *chronological backtracking*.

Nowadays, *Conflict-Driven Clause Learning* (CDCL) SAT solvers are very effective in practice. Their organization is still primarily inspired by DPLL solvers. Moreover, it has been shown that CDCL SAT solvers can also be seen as resolution engines [HBPG08, PD11].

Algorithm 1 shows the standard organization of a CDCL SAT solver first proposed by Marques-

Algorithm 1: CDCL SAT Solver

```
   Input: φ
   Output: satisfying assignment to φ or UNSAT
 1 (dl, conflicts, answer) ← (0, 0, UNKNOWN)
 2 if UnitPropagation(φ) = CONFLICT then
 3 |  answer ← UNSAT
 4 while answer = UNKNOWN do
 5 |  dl ← 0
 6 |  while conflicts < limit ∧ answer = UNKNOWN do
 7 |  |  if AllVariablesAssigned(φ) then
 8 |  |  |  answer ← SAT
 9 |  |  else
10 |  |  |  dl ← dl + 1
11 |  |  |  AssignBranchingVariable(φ)
12 |  |  |  while UnitPropagation(φ) = CONFLICT ∧ answer = UNKNOWN do
13 |  |  |  |  β ← ConflictAnalysis(φ)
14 |  |  |  |  if β < 0 then
15 |  |  |  |  |  answer ← UNSAT
16 |  |  |  |  else
17 |  |  |  |  |  Backtrack(φ, β)
18 |  |  |  |  |  dl ← β
19 |  |  |  |  |  conflicts ← conflicts + 1
20 |  Restart(φ, limit)
21 return answer
```

Silva and Sakallah [MSS96, MSS99]. Since the introduction of CDCL SAT solvers, several techniques have been additionally proposed, namely search restarts [LSZ93, GSK98, BMS00] and implementation of learned clause deletion policies [MSS99, GN02b].

Definition 2.4 (Learned Clause)**.** *Every time a conflict occurs, a new clause is created by analyzing the conflict. The clause that is learned from the conflict is denoted by learned clause.*

Search restarts cause the algorithm to restart itself after a given number of conflicts (*limit*), but clauses already learnt are kept. Since search restarts are now a part of modern CDCL SAT solvers they are also shown in algorithm 1. In addition, learned clause deletion policies are used to decide which learned clauses can be deleted. Learned clause deletion allows the memory usage of the SAT solver to be kept under control. For a more detailed description of CDCL SAT solvers see the work done by Marques-Silva et al. [MSLM09].

In Algorithm 1, the following auxiliary functions are used:

- UnitPropagation consists of the iterated application of the unit clause rule. Given a unit clause, the *unit clause rule* [DP60, DLL62] may be applied: the unassigned literal has to be assigned value 1 for the clause to be satisfied. If an unsatisfied clause is identified, then

conflict is returned.

- AllVariablesAssigned tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable. This is done in modern SAT solvers by checking if there are no more variables to be assigned.

- AssignBranchingVariable consists of choosing a variable to assign and its respective truth value (false or true). Different heuristics have been explored in the past, including the VSIDS (Variable State Independent Decaying Sum) heuristic [MMZ+01] which is considered to be one of the most effective. This heuristic associates an activity counter with each variable. Whenever a learned clause is created after a conflict, each variable that belongs to that clause has its activity increased by a predefined value. Periodically, all activity counters are divided by some experimentally tuned number. When AssignBranchingVariable is called, the highest-value unassigned variable is chosen.

- ConflictAnalysis consists of analyzing the most recent conflict and learning a new clause from the conflict. The learned clause is created by analyzing the structure of unit propagation and deciding which literals to include in the learned clause [BS97, MSS99, MMZ+01, ZMMM01]. If the conflict does not depend on any variable assignment, backtrack cannot be performed and *-1* is returned. Otherwise, it returns the decision level to which the solver can safely backtrack.

- Backtrack backtracks the search to the decision level computed by ConflictAnalysis. Backtracking can be *non-chronological*. While chronological backtracking always backtracks to the last decision level where one of the branching variables values has not been tried, non-chronological backtracking can perform larger backtracks to lower decision levels. While backtracking, all variables at decision levels higher than the one computed by ConflictAnalysis become unassigned.

- Restart consists of restarting the search by unassigning all variables except the ones that have been assigned at decision level 0. Variables that are assigned at decision level 0 correspond to necessary assignments, i.e. these variables can only be assigned by a fixed truth value. After restarting, the cutoff for the next restart is set (*limit*). Rapid randomized restarts are usually used in order to eliminate the heavy-tail behavior [LSZ93, GSK98, BMS00].

Algorithm 1 illustrates how modern SAT solvers work. The algorithm receives a CNF formula φ and applies unit propagation (line 2). If a conflict is found, then the formula is trivially unsatisfiable. Otherwise, the search process begins. The search is done until all variables are assigned a value or until unsatisfiability is proven. At each step, the solver increases the decision level (dl) (line 10). Next, it chooses a variable and its respective value (line 11) and applies unit propagation (line 12). If no conflict is found, then the three steps shown in lines 10 to 12 are repeated. On the other hand, if a conflict is found, then conflict analysis is performed in order to learn a clause that will prevent the same conflict from occurring in the future, and to determine the decision level to which the solver can safely backtrack to (β in line 13). Notice that if β is less than zero, then the conflict does not depend on any variable assignment. As a result, backtracking cannot be performed and the formula is deemed unsatisfiable. Otherwise, backtracking is performed to decision level β and all assignments made at decision levels higher than β are undone (line 17). The decision level is updated to the current backtrack level (line 18) and the number of conflicts is increased by 1 (line 19). The solver returns to line 12 and repeats this process until unsatisfiability has been proven or no more conflicts are found. If the number of conflicts is greater than a given cutoff, then the search is restarted (line 20). When restarting, all variables with decision level higher than 0 are unassigned, the cutoff for the next restart is updated and the decision level is set to 0. After each restart the search begins as usual and it is done until a solution is found or unsatisfiability is proven. For some applications, if unsatisfiability is proven, then a more detailed answer may be needed. In this case, the SAT solver can be extended to obtain an unsatisfiable subformula. These subformulas are denoted by *unsatisfiable cores* and are useful to explain the reason for unsatisfiability. A detailed explanation of how these unsatisfiable cores are obtained can be found in the literature [ZM03, ANORC10].

2.3 Parallel SAT Solving

Nowadays, multicore processors are becoming prevalent. For SAT solvers to improve, it is essential to exploit this new architecture. In the last years, the number of parallel SAT solvers for shared memory architectures has increased significantly. The SAT Race 2008[1] presented the first evaluation of parallel SAT solvers with 3 parallel SAT solvers [HJS09b, CS08, LSB07]. The number of parallel solvers kept on increasing in the last competitions, with 12 parallel solvers in the SAT

[1] `http://baldur.iti.uka.de/sat-race-2008/`

Table 2.1: MANYSAT: different strategies.

	Restart	Heuristic	Polarity	Learning
Thread 1	Geometric $x_1 = 100$ $x_i = 1.5 \times x_{i-1}$	VSIDS (3% rand.)	if $\#occ(x) > \#occ(\bar{x})$ l = true first else l = false first	CDCL (extended [ABH$^+$08])
Thread 2	Dynamic (fast) $\alpha = 1200$ $x_1 = 100, x_2 = 100$ $x_i = f(y_{i-1}, y_i), i > 2$ if $y_{i-1} < y_i$ $f(y_{i-1,y_i}) =$ $\frac{\alpha}{y_i} \times \left\lvert cos(1 - \frac{y_{i-1}}{y_i}) \right\rvert$ else $f(y_{i-1,y_i}) =$ $\frac{\alpha}{y_i} \times \left\lvert cos(1 - \frac{y_i}{y_{i-1}}) \right\rvert$	VSIDS (2% rand.)	Progress saving [PD07]	CDCL
Thread 3	Arithmetic $x_1 = 16000$ $x_i = x_{i-1} + 16000$	VSIDS (2% rand.)	false first	CDCL
Thread 4	Luby 512 [LSZ93]	VSIDS (2% rand.)	Progress saving [PD07]	CDCL (extended [ABH$^+$08])

competition 2011[2] and 19 in the SAT challenge of 2012[3].

There are two main approaches in parallel SAT solving: *portfolios* and *search space splitting*. This section is organized as follows. First, the portfolio approach is described. Afterwards, the search space splitting approach is presented. Finally, other approaches for parallel SAT solving are briefly surveyed. A more detailed description of approaches for parallel SAT solving can be found in a recently published overview by Martins et al. [MML12a].

2.3.1 Portfolios

Portfolios explore the parallelism obtained from running different strategies on the same problem instance. Using complementary sequential SAT algorithms allows to perform as well as the best algorithm for that problem. Moreover, if the algorithms cooperate between themselves, for example by exchanging learned clauses, it is possible to outperform the best strategy for a given problem. The design of a portfolio parallel SAT solver is based on using different parameters for each SAT algorithm. For example, each SAT algorithm can have different restart strategies, decision heuristics, polarity strategies, and learning strategies.

MANYSAT [HJS09b] was the first portfolio parallel SAT solver for a multicore architecture with 4 threads. Table 2.1 shows the different strategies implemented in MANYSAT. For each thread,

[2] `http://www.cril.univ-artois.fr/SAT11/`
[3] `http://baldur.iti.kit.edu/SAT-Challenge-2012/`

the restart heuristic, the variable heuristic, the polarity of variables and the learning scheme differ. The restarts heuristics differ on updating the number of conflicts that are necessary to perform a restart. The variable decision heuristics differ on the percentage of time that a variable is chosen randomly instead of using the order given by the VSIDS heuristic. The heuristics for choosing the polarity of a variable differ on the default truth value assigned to the variables. Finally, the learning schemes used are the common CDCL approach [MSS99, ZMMM01] and an extension of CDCL [ABH+08].

Each thread in ManySAT cooperates by exchanging learned clauses with 8 or less literals. Since the first version of ManySAT, several techniques have been added to this solver in order to improve its performance. For example, the size of learned clauses tends to increase over time, thus reducing the number of learned clauses that are exchanged between the different threads. In order to maintain a steady exchange of learned clauses, one may use dynamic heuristics that adjust the size and the quality of exchanged learned clauses [HJS09a]. Additionally, ManySAT can be improved by using diversification and intensification strategies [GHJS10]. For instance, when considering 4 threads, one can use 2 masters and 2 slaves. The masters use different SAT algorithms to ensure diversification, whereas the slaves intensify the search done by the masters. Moreover, since the master and the slave are searching on similar search spaces, clauses that have been learnt by these threads are expected to be more relevant for each other.

Even though parameter tuning is able to provide complementary SAT algorithms, it may be the case that different sets of parameters perform similarly. If the number of threads is high, it may be difficult to find that many algorithms that provide orthogonal performance. Therefore, the main challenge for portfolio approaches is scalability, since it is difficult to ensure diversification through algorithms that complement each other.

Following the success of ManySAT, new portfolio solvers have emerged in the last years [Bie10, KK11, AHJ+12]. Nowadays, portfolio-based solvers are predominant in parallel SAT solving for multicore architectures.

2.3.2 Search Space Splitting

Search space splitting approaches are based on dividing the search space into disjoint subspaces to be explored in parallel. The most common form of search splitting makes use of guiding paths [ZBH96].

The guiding path consists of a list of decision variables. For each variable, the guiding path

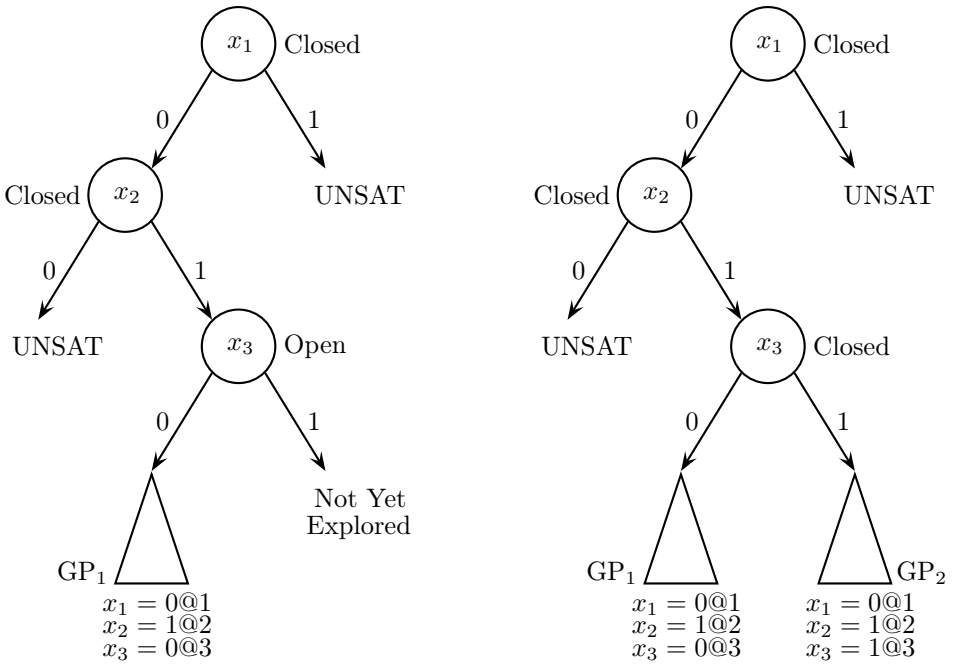


Figure 2.1: Example of division of the search space through the use of guiding paths.

stores the truth value that has been assigned to that variable and a Boolean flag that denotes whether both values have been tried on that variable. A variable where both values have been tried is said to be *closed*. On the other hand, if only one of the values has been tried, the variable is said to be *open*. Open variables represent disjoint subspaces that are still not explored.

Figure 2.1 shows an example of dividing the search space through the use of guiding paths. The left-hand side shows the current search state of thread 1. At decision level 1 the variable x_1 is assigned value 0 ($x_1 = 0@1$), at decision level 2 the variable x_2 is assigned value 1 ($x_2 = 1@2$) and at decision level 3 the variable x_3 is assigned value 0 ($x_3 = 0@3$). The guiding path of thread 1 describes this sequence of assignments and is given by $GP_1 = \langle(x_1, 0, \mathit{closed}),\ (x_2, 1, \mathit{closed}),\ (x_3, 0, \mathit{open})\rangle$. In the guiding path, each variable has the truth value that was assigned to it (0 or 1) and a Boolean flag that shows whether there was an attempt to assign both values (0 and 1) to that variable. For example $(x_1, 0, \mathit{closed})$ shows that x_1 is assigned value 0 and both values were already tried on this variable. On the other hand, $(x_3, 0, \mathit{open})$ shows that x_3 is assigned value 0 but there was still no attempt to assign value 1 to this variable.

The right-hand side shows how the open variables in the guiding path can be used to create disjoint subspaces that can be searched in parallel. For example, thread 2 can start searching on the subspace specified by the guiding path $GP_2 = \langle(x_1, 0, \mathit{closed}),\ (x_2, 1, \mathit{closed}),\ (x_3, 1, \mathit{closed})\rangle$.

Finally, to guarantee that no other thread will work on the same guiding path as GP_2, the guiding path of thread 1 gets modified by marking the previously open decision variable x_3 as *closed*: $GP'_1 = \langle (x_1, 0, closed), (x_2, 1, closed), (x_3, 0, closed) \rangle$.

Even though guiding paths can be used to split the search space, there are some subspaces that are easier to prove to be (un)satisfiable than others. Since the time needed to prove (un)satisfiability on each subspace cannot be predicted, the work cannot be balanced prior to search. Therefore, dynamic work stealing procedures have to exist in order to balance the work between all threads [BS96]. Without such procedure some threads may quickly become idle while others can take a long time to solve their subspace.

Search space splitting solvers are usually based on the master-slave principle. This principle is a model for a communication protocol in which a master thread controls one or more slave threads. The master thread is responsible for maintaining a workload balance between all slave threads. If a slave thread is idle, then it sends a request for a new search space to the master thread. Upon request from a slave thread, if the master thread has some unexplored guiding paths, then it sends a new guiding path to the slave. Otherwise, the master thread selects the slave that has the shortest guiding path (with respect to the number of literals) to split its search space and provide a new unexplored subspace of the search. The shortest guiding path is usually chosen since it corresponds to a larger subspace of the search. Most parallel solvers that use search space splitting follow a variant of this approach [LSB07, CS08, HKWB11].

2.3.3 Other Approaches

Even though search space splitting and portfolio approaches dominate parallel SAT solving, there are a few other relevant approaches, namely, collaborative solving [FS02, VSDK09, DVSK09], parallelization of unit propagation [Man11, HW12], problem splitting [SM08, HMSW11] and hybrid approaches [Blo05, SB10, MML10b].

Similarly to search space splitting approaches, collaborative solvers [FS02, VSDK09, DVSK09] also follow the master-slave principle. However, in collaborative approaches, the master thread runs a sequential SAT algorithm, whereas the slave threads perform parallel inferences that will help the sequential solving. For example, slave threads may extend the guiding path of the master thread by branching on a different decision variable. This results in an unexplored subspace. If a solution was found in that subspace, then the solver can terminate. On the other hand, if that subspace is proven to be unsatisfiable, then the slave thread can send to the master the

backtracking information. Note that, if the master backtracks before any communication of the slave threads, then the slave threads are just restarted with a new guiding path from the master.

Another approach consist on the parallelization of procedures that are called many times by the solver, for example unit propagation. However, in the worst case, unit propagation is inherently sequential [Kas90], i.e. applying unit propagation to a formula may result in a chain of successive and unique implications. Theoretical results are not encouraging and that may explain why unit propagation is usually not parallelized in parallel SAT solvers. However, recently there have been attempts to parallelize unit propagation [Man11, HW12]. For example, the formula can be split into disjoint subformulas which are distributed between the different threads [Man11]. Each thread runs unit propagation on its subformula and then waits for the remaining threads to finish their own unit propagation. The propagation of some variables may imply a given truth value of other variables. Those variables are denoted by *implied variables*. Implied variables that were found during propagation are sent to the remaining threads. Next, each thread propagates the implied variables that were found by all threads. This procedure is done until a fix point is reached. The main challenge of this approach is to partition the formula between the different threads in such a way that the work is evenly balanced. An alternative to split the formula for performing parallel unit propagation, is to partition the variables that need to be propagated, such that each thread propagates a different set of variables [HW12]. As in the previous approach, this procedure is repeated until there are no more variables to propagate. If the variables are partitioned, then all threads need to share the same formula. Therefore, expensive synchronization mechanisms are needed to guarantee the correctness of the parallel unit propagation algorithm.

Another approach for parallel SAT solving is to split the formula into several subformulas that can be solved individually by each thread [SM08, HMSW11]. For example, one may find all solutions for each disjoint subformula [SM08]. Next, if the solutions intersect into a global solution that satisfies all disjoint subformulas, then a solution to the original formula has been found. Otherwise, no global solution exists, and the original formula is unsatisfiable. Alternatively, every time a solution is found for a subformula, one may attempt to extend this solution to satisfy all subformulas [HMSW11]. If this is not possible, then we may extract some information from every unsatisfiable subformula in order to refine each subformula. This approach contrasts with the previous one since computing all solutions of disjoint subformulas may not be required. The main challenge for these approaches is on the overhead induced by merging the information from the subformulas to find a global solution to the original formula.

Finally, hybrid approaches may be created by combining already existing approaches for parallel SAT solving. For example, one may combine search space splitting with portfolio approaches [Blo05, SB10, MML10b]. These hybrid approaches begin with an initial stage of search space splitting and switch to a portfolio approach when load balancing becomes an issue or when a cutoff in the number of conflicts is reached. The main challenge of hybrid approaches is in deciding when to switch from one approach to another one.

2.4 Summary

This chapter described SAT and the main sequential and parallel approaches for SAT solving. CDCL SAT solvers are the most common approach for sequential SAT solving. This chapter described the organization of CDCL SAT solvers and how they can be used to solve a SAT formula. Additionally, the main approaches for parallel SAT solving were also surveyed in this chapter, namely, portfolios and search space splitting. Moreover, other approaches for parallel SAT solving, such as, collaborative solving, parallelization of unit propagation, problem splitting and hybrid approaches were also briefly surveyed in this chapter.

3

Maximum Satisfiability

Maximum Satisfiability (MaxSAT) is an optimization version of Boolean Satisfiability (SAT) where the goal is to find an assignment to the problem variables such that the number of satisfied clauses is maximized. Recently, new algorithms have been proposed [FM06, MMSP09, ABL09, ABL10a, HMMS11, AZFH12], which are particularly effective for solving problem instances encoding real-world problems. As a result of these algorithmic advances, MaxSAT solvers are nowadays powerful tools that can be used in many important application domains, such as software package upgrades [ABL+10b], error localization in C code [JM11], debugging of hardware designs [CSMSV10], haplotyping with pedigrees [GLMSO10], and course timetabling [AN12].

This chapter is organized as follows. First, we provide some MaxSAT definitions that will be used throughout this dissertation. Next, we present the main algorithmic solutions for MaxSAT. Afterwards, we briefly describe encodings that translate cardinality constraints into clauses. These encodings will be used in the MaxSAT algorithms presented in the remainder of the dissertation. Finally, an experimental evaluation of the impact of different encodings in the performance of MaxSAT algorithms is presented.

3.1 Definitions

Definition 3.1 (Maximum Satisfiability)**.** *Given a CNF formula φ, the Maximum Satisfiability (MaxSAT) problem can be defined as finding an assignment to variables in φ that minimizes (maximizes) the number of unsatisfied (satisfied) clauses. Such assignment is called an optimal solution.*

MaxSAT has several variants such as partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT. In the partial MaxSAT problem, some clauses in φ are declared as hard, while the rest are declared as soft. The objective in partial MaxSAT is to find an assignment to problem variables such that all hard clauses are satisfied, while minimizing the number of unsatisfied soft clauses. Finally, in the weighted versions of MaxSAT, soft clauses can have weights greater than 1 and the objective is to satisfy all hard clauses while minimizing the total weight of unsatisfied soft clauses.

Example 3.1. *Consider a partial MaxSAT formula φ such that $\varphi = \varphi_h \cup \varphi_s$, where φ_h denotes the set of hard clauses and φ_s the multiset of soft clauses. Furthermore, consider that*

$$\begin{aligned} \varphi_h = &\quad \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3]\} \\ \varphi_s = &\quad \{(x_1), (x_2 \vee \bar{x}_1), (x_3), (\bar{x}_3 \vee x_1)\} \end{aligned} \tag{3.1}$$

Note that hard clauses are represented between square brackets while soft clauses are represented between round brackets. An example of an optimal assignment is $\nu = \langle x_1, \bar{x}_2, \bar{x}_3 \rangle$. This assignment satisfies all hard clauses and only two soft clauses are unsatisfied.

Example 3.2. *Consider a weighted partial MaxSAT formula $\varphi = \varphi_h \cup \varphi_s$ such that:*

$$\begin{aligned} \varphi_h = &\quad \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3]\} \\ \varphi_s = &\quad \{(x_1, 3), (x_2 \vee \bar{x}_1, 1), (x_3, 2), (\bar{x}_3 \vee x_1, 1)\} \end{aligned} \tag{3.2}$$

A soft clause ω with weight w is represented as (ω, w). An example of an optimal assignment is $\nu = \langle x_1, \bar{x}_2, \bar{x}_3 \rangle$. This assignment satisfies all hard clauses while minimizing the total weight of unsatisfied soft clauses, i.e. only the soft clauses $(x_2 \vee \bar{x}_1, 1), (x_3, 2)$ are unsatisfied, which corresponds to an optimal solution with value 3.

A generalization of clauses are cardinality constraints. Although cardinality constraints do not occur in MaxSAT formulations, several algorithms for MaxSAT rely on these constraints [ABL09, HMMS11, MML12e].

Definition 3.2 (Cardinality Constraints)**.** *Consider n literals and a given value k, such that $k \leq n$. Cardinality constraints ensure that at least k literals are assigned truth value 1. More formally, cardinality constrains define that a sum of n literals must be greater than or equal to a given value k, i.e. $\sum_{i=1}^{n} l_i \geq k$. However, in practice cardinality constraints are usually expressed as* at-most-k *constraints. Note that the previous* at-least-k *expression can be rewritten as $\sum_{i=1}^{n} \bar{l}_i \leq n-k$. In the remainder of the dissertation, cardinality constraints are defined as being* at-most-k *constraints.*

A special case of cardinality constraints are the at-most-one *constraints. These constraints are defined for $k = 1$, and express that at most one out of n literals can be assigned truth value 1.*

Cardinality constraints can be generalized into pseudo-Boolean constraints.

Definition 3.3 (Pseudo-Boolean Constraints)**.** *Literals in cardinality constraints have coefficient 1. However, if the coefficients are larger than 1, then these constraints are denoted as pseudo-Boolean constraints. More formally, a pseudo-Boolean constraint is defined as a linear inequality over a set of n literals of the following form:*

$$\sum_{j=1}^{n} a_j l_j \leq k, \text{ such that for each } j \in \{1 \dots n\}, a_j, k \in \mathbb{Z}^+. \tag{3.3}$$

Pseudo-Boolean constraints do not occur in MaxSAT formulations but are used by several weighted partial MaxSAT algorithms [MMSP09, ABL10a, HMMS11].

Example 3.3. *An example of a cardinality constraint is $x_1 + x_2 + \bar{x}_3 + x_4 \leq 2$. The assignment $\nu = \langle x_1, x_2, x_3, \bar{x_4} \rangle$ satisfies the cardinality constraint since there are only two literals that are assigned truth value 1. On the other hand, the assignment $\nu' = \langle x_1, x_2, \bar{x}_3, \bar{x}_4 \rangle$ unsatisfies the cardinality constraint since there are three literals that are assigned truth value 1. An example of a pseudo-Boolean constraint is $3x_1 + x_2 + 2x_3 + x_4 \leq 2$. The assignment $\nu = \langle \bar{x}_1, x_2, \bar{x}_3, x_4 \rangle$ satisfies the pseudo-Boolean constraint, whereas the assignment $\nu' = \langle x_1, \bar{x}_2, \bar{x}_3, \bar{x}_4 \rangle$ unsatisfies the pseudo-Boolean constraint.*

3.2 MaxSAT Algorithms

In the last decade, several algorithms for solving MaxSAT have been proposed. This section surveys MaxSAT algorithms that are used in this dissertation, namely linear search algorithms [BP10, AZFH12] and unsatisfiability-based algorithms [FM06, MMSP09, ABL09]. Moreover, other MaxSAT algorithms are also briefly described.

3.2.1 Linear Search Algorithms

One approach for solving MaxSAT is to make a linear search on the number of unsatisfied soft clauses. Algorithm 2 shows the traditional linear search algorithm for partial MaxSAT [BP10, AZFH12].

Algorithm 2: Linear search algorithm for partial MaxSAT

Input: $\varphi = \varphi_h \cup \varphi_s$
Output: satisfying assignment to φ or UNSAT

```
1  (V_R, model, μ, φ_W) ← (∅, ∅, +∞, φ_h)
2  foreach ω ∈ φ_s do
3      V_R ← V_R ∪ {r}                                   // r is a new variable
4      ω_R ← ω ∪ {r}                                     // relax soft clause
5      φ_W ← φ_W ∪ ω_R
6  while true do
7      (st, ν, φ_C) ← SAT(φ_W)
8      if st = SAT then
9          model ← ν
10         μ ← |{r ∈ V_R | ν(r) = 1}|                    // number of r variables assigned to 1
11         φ_W ← φ_W ∪ {CNF(Σ_{r∈V_R} r ≤ μ − 1)}
12     else
13         if model = ∅ then
14             return UNSAT                              // the MaxSAT formula is unsatisfiable
15         else
16             return model                              // return satisfying assignment to φ_W
```

The algorithm starts by relaxing the partial MaxSAT formula. For each soft clause ω, a new variable is created and added to ω (lines 2-3). Next, the relaxed soft clause is added to the working formula φ_W. The goal is to find an assignment to the problem variables such that it minimizes the number of relaxation variables that are assigned truth value 1. If a relaxation variable is assigned truth value 1, then it corresponds to the unsatisfiability of a soft clause in the original partial MaxSAT formula.

Linear search algorithms work by iteratively calling a SAT solver over a working formula φ_W. A SAT solver returns a triple (st, ν, φ_C), where st denotes the outcome of the solver: satisfiable or unsatisfiable. If the solver returns satisfiable, then the model that satisfies all clauses is stored in ν. On the other hand, if the solver returns unsatisfiable, then φ_C contains an unsatisfiable core.

The working formula φ_W is then given to the SAT solver. While the working formula remains satisfiable, the model ν provided by the SAT solver is stored and we compute the upper bound μ corresponding to the model ν. This upper bound gives the current number of soft clauses that are unsatisfied in the original partial MaxSAT formula and corresponds to the number of relaxation variables that are assigned truth value 1 (line 10). Next, the working formula φ_W is updated by adding a cardinality constraint that excludes models with a value greater than or equal to μ.

This procedure is repeated until the SAT solver returns unsatisfiable. When this occurs, an optimal solution to φ has been found (line 16). If there was no satisfiable call to the SAT solver, then the original formula is unsatisfiable (line 14).

In practice, cardinality constraints are usually encoded into CNF [BB03, Sin05, ES06] so that a SAT solver can be used. Moreover, for several cardinality encodings we do not need to re-encode the cardinality constraint at each SAT call. Since the upper bound value is always decreasing, it is possible to update the CNF representation of the cardinality constraint by setting some specific literals to false. This procedure is denoted by *incremental strengthening* [ANORC11].

Linear search algorithms perform an *upper bound search* on the value of the optimal solution, i.e. at each moment of the search process these algorithms maintain a candidate solution that is iteratively improved until optimally is proved. The candidate solutions always have a value that is greater than or equal to the optimum.

Example 3.4. *Consider the partial MaxSAT formula φ as defined in Equation (3.1). Algorithm 2 starts relaxing the partial MaxSAT formula by introducing a new relaxation variable in each soft clause. As a result, the working formula φ_W will be updated as follows:*

$$\begin{aligned} \varphi_W = \quad & \{(\bar{x}_2 \vee \bar{x}_1), (x_2 \vee \bar{x}_3), \\ & (x_1 \vee r_1), (x_2 \vee \bar{x}_1 \vee r_2), (x_3 \vee r_3), (\bar{x}_3 \vee x_1 \vee r_4)\} \end{aligned} \tag{3.4}$$

The set of relaxation variables is $V_R = \{r_1, r_2, r_3, r_4\}$. Next, φ_W is given to a SAT solver. Suppose that the SAT solver returns the following satisfying assignment $\nu = \langle \bar{x}_1, x_2, \bar{x}_3, r_1, \bar{r}_2, r_3, \bar{r}_4 \rangle$. We store ν in model *and compute the upper bound μ. Since r_1 and r_3 were assigned truth value 1, then we have found a solution that unsatisfies two soft clauses. Therefore, we can update the upper bound μ to 2.*

The working formula is now updated with a cardinality constraint over all relaxation variables, such that assignments corresponding to the unsatisfiability of two or more soft clauses are excluded. As a result, the working formula is now as follows:

$$\begin{aligned} \varphi_W = \quad & \{(\bar{x}_2 \vee \bar{x}_1), (x_2 \vee \bar{x}_3), \\ & (x_1 \vee r_1), (x_2 \vee \bar{x}_1 \vee r_2), (x_3 \vee r_3), (\bar{x}_3 \vee x_1 \vee r_4), \\ & \mathit{CNF}(r_1 + r_2 + r_3 + r_4 \leq 1)\} \end{aligned} \tag{3.5}$$

After updating the working formula, the algorithm makes another call to the SAT solver which returns unsatisfiable. This means that there is no satisfying assignment such that only one of the relaxation variables is assigned truth value 1. Therefore, the optimal solution is given by the previous model that corresponds to unsatisfying two soft clauses.

Algorithm 2 can be easily modified for solving weighted partial MaxSAT problems. The dif-

ference between the weighted and unweighted versions lies on the computation of μ and on the constraint that is added on the relaxation variables to exclude solutions with a greater or equal value to μ. For weighted partial MaxSAT instances, the weights of soft clauses may be larger than 1. Therefore, if a relaxation variable is assigned truth value 1, then it will increase μ by the weight of the respective soft clause. This contrasts with the unweighted version, where the assignment of truth value 1 to a relaxation variable increases μ by 1. Due to the weight of soft clauses, a pseudo-Boolean constraint is added on the relaxation variables to exclude solutions with a value greater or equal to μ. The coefficients of the literals in the pseudo-Boolean constraint correspond to the weight of the soft clauses.

Example 3.5. *Consider the weighted partial MaxSAT formula φ as defined in Equation (3.2). The weighted version of algorithm 2 starts by relaxing the weighted partial MaxSAT formula in a similar way to the partial MaxSAT formula. As a result, the working formula φ_W will be updated as follows:*

$$\begin{aligned} \varphi_W = \quad & \{(\bar{x}_2 \vee \bar{x}_1), (x_2 \vee \bar{x}_3), \\ & (x_1 \vee r_1, 3), (x_2 \vee \bar{x}_1 \vee r_2, 1), (x_3 \vee r_3, 2), (\bar{x}_3 \vee x_1 \vee r_4, 1)\} \end{aligned} \tag{3.6}$$

Suppose the SAT solver returns the following satisfying assignment $\nu = \langle x_1, \bar{x}_2, \bar{x}_3, \bar{r}_1, r_2, r_3, \bar{r}_4 \rangle$. Since r_2 and r_3 were assigned truth value 1, then we have found a solution that unsatisfies two soft clauses with weights 1 and 2. Therefore, we can update the upper bound μ to 3, which is the sum of the weights of the unsatisfied soft clauses.

The working formula is now updated with a pseudo-Boolean constraint over all relaxation variables, such that assignments corresponding to the unsatisfiability of soft clauses with a sum of weights greater than or equal to 3 are excluded. As a result, the working formula is now as follows:

$$\begin{aligned} \varphi_W = \quad & \{(\bar{x}_2 \vee \bar{x}_1), (x_2 \vee \bar{x}_3), \\ & (x_1 \vee r_1, 3), (x_2 \vee \bar{x}_1 \vee r_2, 1), (x_3 \vee r_3, 2), (\bar{x}_3 \vee x_1 \vee r_4, 1)\} \\ & \mathit{CNF}(3r_1 + r_2 + 2r_3 + r_4 \leq 2)\} \end{aligned} \tag{3.7}$$

After updating the working formula, the algorithm makes another call to the SAT solver which returns unsatisfiable. Therefore, an optimal solution was found and it is given by the last found model.

Algorithm 3: Unsatisfiability-based algorithm for partial MaxSAT

Input: $\varphi = \varphi_h \cup \varphi_s$
Output: satisfying assignment to φ or UNSAT

```
1  (st, ν, φ_C) ← SAT(φ_h)                    // check if the MaxSAT formula is unsatisfiable
2  if st = UNSAT then
3      return UNSAT
4  φ_W ← φ
5  while true do
6      (st, ν, φ_C) ← SAT(φ_W)
7      if st = UNSAT then
8          V_R ← ∅
9          foreach ω ∈ (φ_C ∩ φ_s) do
10             V_R ← V_R ∪ {r}                          // r is a new variable
11             ω_R ← ω ∪ {r}                            // relax soft clause
12             φ_W ← φ_W \ {ω} ∪ {ω_R}
13         φ_W ← φ_W ∪ {CNF(Σ_{r∈V_R} r ≤ 1)}
14     else
15         return ν                                     // satisfying assignment to φ_W
```

3.2.2 Unsatisfiability-based Algorithms

Recently, a new generation of MaxSAT solvers have been developed based on the iterated use of a SAT solver to identify unsatisfiable cores [FM06, MMSP09, ABL09]. Algorithm 3 presents the pseudo-code for Fu and Malik's proposal [FM06].

Consider a partial MaxSAT instance φ, where clauses are marked as either soft or hard. Algorithm 3 starts by checking if φ is satisfiable by calling a SAT solver only with hard clauses φ_h. If φ_h is satisfiable, then the working formula φ_W is initialized with φ. Otherwise, if φ_h is unsatisfiable, then φ is unsatisfiable and the algorithm terminates. At each iteration, a SAT solver is used. If the solver returns unsatisfiable, then each soft clause in φ_C is relaxed (lines 10-12). This relaxation procedure consists in adding a new relaxation variable to each soft clause in φ_C. Moreover, φ_W is changed to encode that at most one of the new relaxation variables can be assigned value 1 (line 13) and the algorithm continues to the next iteration. Otherwise, if φ is satisfiable, then the SAT solver was able to find an assignment which is an optimal solution to the original partial MaxSAT problem [FM06].

Example 3.6. *Consider again the partial MaxSAT formula φ as defined in Equation (3.1). If all clauses are given to the SAT solver, then the formula is clearly unsatisfiable and an unsatisfiable core φ_C is identified. Suppose that $\varphi_C = \{[\bar{x}_2 \vee \bar{x}_1], (x_1), (x_2 \vee \bar{x}_1)\}$. In this case, a new relaxation variable would be added to each soft clause in φ_C, as well as a new constraint restricting that at most one of the relaxation variables can be assigned truth value 1. As a result, the formula would*

be updated as follows:

$$\begin{aligned}\varphi_W = \quad & \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], \\ & (x_1 \vee r_1), (x_2 \vee \bar{x}_1 \vee r_2), (x_3), (\bar{x}_3 \vee x_1), \\ & [\mathit{CNF}(r_1 + r_2 \leq 1)]\}\end{aligned} \tag{3.8}$$

Notice that the at-most-one constraint must be encoded into CNF in order to be able to continue using a SAT solver in the next iteration. Since the formula is still unsatisfiable, a new unsatisfiable core is identified. Suppose we now have $\varphi_C = \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], (x_3), (\bar{x}_3 \vee x_1)\}$*. In this case, the working formula would be updated as follows:*

$$\begin{aligned}\varphi_W = \quad & \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], \\ & (x_1 \vee r_1), (x_2 \vee \bar{x}_1 \vee r_2), (x_3 \vee r_3), (\bar{x}_3 \vee x_1 \vee r_4), \\ & [\mathit{CNF}(r_1 + r_2 \leq 1)], [\mathit{CNF}(r_3 + r_4 \leq 1)]\}\end{aligned} \tag{3.9}$$

The resulting formula is now satisfiable. Consider that the SAT solver returns the satisfying assignment $\nu = \langle x_1, \bar{x}_2, \bar{x}_3, \bar{r}_1, r_2, r_3, \bar{r}_4 \rangle$*. In this example, the optimal solution has a value of two, corresponding to two unsatisfiable soft clauses. Notice that the value of a optimal solution of a partial MaxSAT instance corresponds to the number of unsatisfiable cores found during the execution of the algorithm [FM06].*

Algorithm 3 can only solve partial MaxSAT problems. However, it is possible to solve weighted partial MaxSAT problems using Algorithm 3. Consider a weighted partial MaxSAT formula φ. For each soft clause (ω, w) in φ, we may replace ω with w copies of weight 1. With this naive procedure, we may transform a weighted partial MaxSAT formula into a partial MaxSAT formula. However, if the weights of soft clauses are large, then it is unlikely that this approach will scale up. Therefore, more effective solutions have been proposed for weighted problems [ABL09, MMSP09].

Algorithm 4 presents an unsatisfiability-based algorithm for a weighted partial MaxSAT formula φ. Similarly to the unweighted version, Algorithm 4 starts by checking if φ_h is satisfiable. If this is the case, then the working formula φ_W is initialized with φ. If φ_h is unsatisfiable, then φ is unsatisfiable and the algorithm terminates. At each iteration, a SAT solver is used. If the solver returns unsatisfiable, then the minimum weight min_c of the unsatisfiable core φ_C is computed (line 11). The *minimum weight* of φ_C corresponds to the smallest weight of the soft clauses in φ_C. For each soft clause (ω, w) in φ_C, if w is equal to min_c, then a new relaxation variable is added to ω (line 14) and ω is updated in the formula (line 20). However, if w is greater than min_c,

Algorithm 4: Unsatisfiability-based algorithm for weighted partial MaxSAT

Input: $\varphi = \varphi_h \cup \varphi_s$
Output: satisfying assignment to φ or UNSAT

1 $(\mathsf{st}, \nu, \varphi_C) \leftarrow \mathtt{SAT}(\varphi_h)$ `// check if the MaxSAT formula is unsatisfiable`
2 **if** st = UNSAT **then**
3 **return** UNSAT
4 $\varphi_W \leftarrow \varphi$
5 **while** true **do**
6 $(\mathsf{st}, \nu, \varphi_C) \leftarrow \mathtt{SAT}(\varphi_W)$
7 **if** st = UNSAT **then**
8 $min_c \leftarrow +\infty$
9 **foreach** $(\omega, w) \in (\varphi_C \cap \varphi_s)$ **do**
10 **if** $w < min_c$ **then**
11 $min_c \leftarrow w$ `// minimum weight of` φ_C
12 $V_R \leftarrow \emptyset$
13 **foreach** $(\omega, w) \in (\varphi_C \cap \varphi_s)$ **do**
14 $V_R \leftarrow V_R \cup \{r\}$ `// r is a new variable`
15 $(\omega_R, w') \leftarrow (\omega \cup \{r\}, min_c)$ `// relax soft clause`
16 **if** $w > min_c$ **then**
17 $(\omega, w) \leftarrow (\omega, w - min_c)$ `// duplicate soft clause`
18 $\varphi_W \leftarrow \varphi_W \cup \{\omega_R\}$
19 **else**
20 $\varphi_W \leftarrow \varphi_W \setminus \{\omega\} \cup \{\omega_R\}$
21 $\varphi_W \leftarrow \varphi_W \cup \{\mathtt{CNF}(\sum_{r \in V_R} r \leq 1)\}$
22 **else**
23 **return** ν `// satisfying assignment to` φ_W

then ω is duplicated into two clauses. One of the clauses is the relaxation of ω with weight min_c (line 15). The other clause is ω with an updated weight of w minus min_c (line 17). The idea is to duplicate soft clauses only when they appear in an unsatisfiable core and they have a weight greater than the minimum weight of the unsatisfiable core. This contrasts with the duplication of all soft clauses given by the naive approach of transforming a weighted partial MaxSAT formula into a partial MaxSAT formula.

Next, φ_W is changed to encode that at most one of the new relaxation variables can be assigned value 1 (line 21) and the algorithm continues to the next iteration. Otherwise, if φ_W is satisfiable, then that means that the SAT solver was able to find an assignment which is an optimal solution to the original weighted MaxSAT problem.

Notice that unsatisfiability-based algorithms perform a *lower bound search*, i.e. at each step of the algorithm, the sum of the minimum weight of the identified unsatisfiable cores provides a lower bound on the value of an optimal solution. These algorithms are complementary to the linear search algorithms described in the previous section.

Example 3.7. *Consider again the weighted partial MaxSAT formula φ as defined in Equation (3.2). If all clauses are given to the SAT solver, then the formula is clearly unsatisfiable and an unsatisfiable core φ_C is identified. Suppose that $\varphi_C = \{[\bar{x}_2 \vee \bar{x}_1], (x_1, 3), (x_2 \vee \bar{x}_1, 1)\}$. In this case, the minimum weight min_c of φ_C is 1. Since $(x_1, 3)$ has a weight larger than min_c, then the clause is duplicated into $(x_1 \vee r_1, 1)$ and $(x_1, 2)$. On the other hand, $(x_2 \vee \bar{x}_1, 1)$ has weight equal to min_c, and therefore it is only relaxed by adding a new relaxation variable to this soft clause. Moreover, a new constraint restricting that at most one of the relaxation variables can be assigned truth value 1 is also added to φ_W. As a result, the formula would be updated as follows:*

$$\begin{aligned} \varphi_W = \quad & \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], \\ & (x_1, 2), (x_1 \vee r_1, 1), (x_2 \vee \bar{x}_1 \vee r_2, 1), (x_3, 2), (\bar{x}_3 \vee x_1, 1), \\ & [\mathit{CNF}(r_1 + r_2 \leq 1)]\} \end{aligned} \tag{3.10}$$

Since the formula is still unsatisfiable, a new unsatisfiable core is identified. Suppose we now have $\varphi_C = \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], (x_3, 2), (x_1, 2)\}$. In this case, the min_c of φ_C is 2. Since all soft clauses in φ_C have weight equal to min_c, the soft clauses are relaxed by adding a new relaxation variable to each soft clause. As a result, the formula is updated as follows:

$$\begin{aligned} \varphi_W = \quad & \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], \\ & (x_1 \vee r_3, 2), (x_1 \vee r_1, 1), (x_2 \vee \bar{x}_1 \vee r_2, 1), (x_3 \vee r_4, 2), (\bar{x}_3 \vee x_1, 1), \\ & [\mathit{CNF}(r_1 + r_2 \leq 1)], [\mathit{CNF}(r_3 + r_4 \leq 1)]\} \end{aligned} \tag{3.11}$$

The resulting formula is now satisfiable. Consider that the SAT solver returns the satisfying assignment $\nu = \langle x_1, \bar{x}_2, \bar{x}_3, \bar{r}_1, r_2, \bar{r}_3, r_4 \rangle$. In this example, the optimal solution has a value of three, corresponding to the sum of the weights of the unsatisfied soft clauses.

Algorithms 3 and 4 present the unsatisfiability-based algorithms used in this dissertation. We refer to the literature for the several variants of unsatisfiability-based algorithms that have been proposed in the last years [MMSP09, ABL09, ABL10a, MML10a, HMMS11, ABL13]. For example, algorithms may differ on the CNF encodings used for the at-most-one constraints [MML11a] or on the relaxation process [ABL09, ABL10a].

3.2.3 Other Algorithmic Solutions

The classical approach to solve MaxSAT is to use a branch and bound algorithm [ALM07, LS07, HLO08, LMP07]. These algorithms keep an upper bound and a lower bound on the optimal value

of the MaxSAT solution. The upper bound value is updated whenever a better solution is found, whereas the lower bound value is estimated considering a set of variables assignments. Whenever the lower bound value is greater than or equal to the upper bound value, the search procedure can safely backtrack since it is guaranteed that the current best solution cannot be improved by extending the current set of variable assignments.

Another approach is to perform conversions of one Boolean formalism to another and subsequently use a specific solver on the new formalism [ES06, BP10, LHG08]. For instance, a MaxSAT instance can be converted into an equivalent pseudo-Boolean optimization (PBO) formula and PBO solvers can be used to solve it [BP10]. Alternatively, MaxSAT instances can also be converted into a weighted constraint network and a Constraint Satisfaction Problem (CSP) solver can be used to solve it [LHG08].

Another alternative approach to solve MaxSAT is to use binary search by updating both lower and upper bound values of the optimal solution [HMMS11, AKFH11]. At each iteration, the algorithm tries to find a solution with a value smaller than or equal to half of the upper bound value. If no solution exists, then the lower bound can be updated. If a solution is found, then the upper bound is updated and the algorithm restarts this procedure. An optimal solution is found when the lower bound is the same as the upper bound. Although one can devise other ways of using binary search schemes, we refer to its use in linear search algorithms [AKFH11], as well as integrated with unsatisfiability-based algorithms [HMMS11].

3.3 Encodings for Cardinality Constraints

Cardinality constraints can arise when solving a MaxSAT formula. In particular, *at-most-k* cardinality constraints are used in the linear search algorithm presented for partial MaxSAT, whereas *at-most-one* cardinality constraints are used in the unsatisfiability-based algorithms presented for partial and weighted partial MaxSAT. If the solver is not able to natively handle cardinality constraints, then it is necessary to translate cardinality constraints into clauses. Several encodings that translate cardinality constraints into clauses have been proposed. In this section, a brief description of several encodings that will be used in the remainder of the dissertation is provided. Moreover, a dynamic heuristic for selecting the encoding for cardinality constraints is also presented. Given a portfolio of encodings and a cardinality constraint, the dynamic heuristic tries to select the most adequate encoding for that constraint.

Our focus is on encodings for cardinality constraints (in the sequel referred as *cardinality*

Table 3.1: Encodings for cardinality constraints

Encoding	#Clauses	#Variables	Type
Pairwise	$\mathcal{O}(n^2)$	0	*at-most-one*
Ladder	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Bitwise	$\mathcal{O}(n\ log_2\ n)$	$\mathcal{O}(log_2\ n)$	*at-most-one*
Commander	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Product	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Sequential	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	*at-most-k*
Totalizer	$\mathcal{O}(nk)$	$\mathcal{O}(n\ log_2\ n)$	*at-most-k*
Sorters	$\mathcal{O}(n\ log_2^2\ n)$	$\mathcal{O}(n\ log_2^2\ n)$	*at-most-k*

encodings) that allow unit propagation to maintain arc consistency in the resulting CNF encoding. Consider the following cardinality constraint $x_1 + \ldots + x_n \leq k$. If k variables are assigned truth value 1, then unit propagation will enforce the truth value 0 on the remaining $n - k$ variables. However, if $k+1$ variables are assigned truth value 1, then a conflict arises since at most k variables can be assigned truth value 1. An encoding that maintains arc consistency enables the SAT solver to infer the same information with the use of unit propagation on the resulting CNF encoding.

Note that one may also translate pseudo-Boolean constraints into clauses. However, encoding pseudo-Boolean constraints into clauses may require a very large number of clauses and variables. On the other hand, cardinality constraints have efficient translations that can be handled by the SAT solver. In the remainder of this dissertation we will restrict ourselves to encodings of cardinality constraints.

A large number of encodings have been proposed to handle *at-most-one* constraints. Therefore, a distinction is made between encodings that are used only for *at-most-one* constraints and encodings used for the general case of *at-most-k* constraints. Table 3.1 shows the size of the evaluated encodings. For each encoding, the number of additional clauses and variables needed to encode the cardinality constraint $x_1 + \ldots + x_n \leq k$ into clauses is presented. For example, the *pairwise* encoding requires a quadratic number of additional clauses to encode an *at-most-one* cardinality constraint, whereas the *ladder* encoding requires a linear number of additional clauses and variables to encode the same *at-most-one* constraint.

Next, the encodings are briefly described. For further details on each encoding, the reader is pointed to the literature.

At-Most-One Cardinality Constraints

- Pairwise (also called naive): the most widely known encoding for the *at-most-one* constraint. For each pair of variables $\langle x_i, x_j \rangle$, add a binary clause $(\bar{x}_i \vee \bar{x}_j)$ that guarantees that only one of the two variables can be assigned truth value 1. Even though this encoding adds a quadratic number of clauses, it does not require auxiliary variables.

- Ladder [GN04, AM04]: it uses $n-1$ auxiliary variables to form a structure named ladder. Consider the chain of auxiliary variables $y_1, \ldots, y_{n+1}$. If y_i is assigned truth value 0 then all variables y_j with $j > i$ are also assigned truth value 0. Each valid state in the ladder is associated with a variable of the cardinality constraint. Since each x_i is equivalent to a valid state in the ladder, this encoding guarantees that at most one variable x_i will be assigned truth value 1.

- Bitwise [FPDN05, Pre07]: this encoding introduces auxiliary variables $y_1, \ldots, y_{log_2 n}$ that represent a bit string. It then associates a unique bit string with each variable x_i. The encoding guarantees that only one string may occur and therefore at most one variable x_i can be assigned truth value 1. When n is not a power of 2, we can perform a small optimization by reducing the number of clauses from the encoding [FPDN05]. Note that if n is not a power of 2, then there are more strings than variables x_i. Hence, we can associate two strings to some of the variables x_i until the number of remaining strings is equal to the number of remaining variables x_i.

- Commander [KK07]: it starts by partitioning the set of variables x_i into groups of size 3. Next, for the variables of each group, an *at-most-one* constraint is encoded with the pairwise encoding. Finally, it associates a commander variable with each group and recursively encodes the *at-most-one* constraint over the commander variables with the method just described.

- Product [Che10]: this encoding decomposes cardinality constraint $x_1 + \ldots + x_n \leq 1$ into two constraints, $y_1 + \ldots + y_{p_1} \leq 1$ and $z_1 + \ldots + z_{p_2} \leq 1$, where $p_1 \times p_2 \geq n$. The idea is to associate each variable x_i with a coordinate (y_a, z_b). This procedure is applied recursively until the size of the constraint is smaller than 7. At that point, the pairwise encoding is used.

At-Most-k Cardinality Constraints

- Sequential [Sin05]: it encodes a circuit that sequentially counts the number of variables x_i that are assigned truth value 1. Each x_i is associated with k variables $s_{i,j}$ that are used as a counter. Assigning the truth value 0 to $s_{i,j}$ implies that at most j of the variables $x_1, \ldots, x_{i-1}$ can be assigned truth value 1.

- Totalizer [BB03]: it consists of a totalizer and a comparator. The totalizer can be seen as a binary tree, where the leaves are the x_i variables. Each intermediate node is labeled with a number s and uses s auxiliary variables to represent the sum of the leaves of the corresponding subtree. The original encoding uses $\mathcal{O}(n^2)$ clauses. However, it is possible to reduce the number of clauses to $\mathcal{O}(nk)$ [BR05]. This optimization consists in counting up to $k+1$, instead of counting up to n.

- Sorters [ES06]: it is based on a sorting network, i.e. a circuit that receives n Boolean inputs $x_1, \ldots, x_n$ and permutes them to obtain the sorted outputs $y_1, \ldots, y_n$. Consider the cardinality constraint $x_1 + \ldots + x_n \leq k$. If after building the sorting network we assign truth value 0 to the output y_{k+1}, then this guarantees that at most k variables x_i can be assigned truth value 1. Some improvements were introduced over the original sorting network encoding, namely, the use of half sorting networks [ANORC11] and adding redundant clauses over the outputs that amplify propagation [CZI10]. Even though the size of the sorters encodings grows with n, unit propagation on the outputs $y_{k+1}, \ldots, y_n$ will significantly reduce the size of the encoding. Therefore, if k is small, then the sorter encoding will be much smaller after unit propagation. Moreover, for the *at-most-one* constraints, the simplification of the sorting network through partial evaluation [CZI10] is used and the size of the encoding is reduced to $\mathcal{O}(n)$ clauses and variables.

3.3.1 Dynamic Encoding Heuristic

For the *at-most-k* cardinality constraint it has been empirically observed that several features may be used to build a dynamic heuristic for selecting the more adequate encoding for each cardinality constraint [MML11a]. Consider the following portfolio of encodings: *totalizer*, *sorters* and the native representation of cardinality constraints, i.e. without encoding them into CNF and using a pseudo-Boolean solver instead of a SAT solver. For a given MaxSAT formula with v variables and an *at-most-k* cardinality constraint with size n, the dynamic encoding heuristic behaves as

follows:

1. If *(i)* $0.25 \leq k/n \leq 0.75$, *(ii)* $n > 1024$ and *(iii)* $n/v < 0.75$:

   - then do not encode the cardinality constraint into clauses. In this case the native representation is used and we rely on a solver that can handle pseudo-Boolean constraints natively.

2. Else if $n \times k < n \times log_2^2\ n$:

   - then encode the cardinality constraint into clauses using the *totalizer* encoding.

3. Otherwise:

   - the cardinality constraint is translated into clauses using the *sorters* encoding.

The native representation is used when the ratio between the number of variables and the value of k is close to $n/2$. This is the worst case when using a CNF representation. However, this is only used when the number of variables in the cardinality constraint is larger than 1024. When n is small, encoding the cardinality constraint into clauses is still more effective than using a native representation, even when k is close to $n/2$. Note that when $k > n/2$ the *at-most-k* constraint can be rewritten as an *at-least-(n-k)* constraint. Let $x_1 + \ldots + x_n \leq k$ be an *at-most-k* constraint. This constraint can be rewritten as $\bar{x}_1 + \ldots + \bar{x}_n \geq n - k$. The *totalizer* and *sorters* encodings allow the encoding of *at-least-(n-k)* constraints. Therefore, if $k > n/2$, the *at-most-k* constraint is rewritten as an *at-least-(n-k)* constraint and then encoded into clauses. Hence, the worst case when using a CNF representation is when k is close to $n/2$ since rewriting the constraint does not reduce the size of the encoding.

It was also observed that when the cardinality constraint contains the majority of the variables of the problem, encoding the cardinality constraint into clauses may lead to better results. The choice between the *totalizer* and *sorters* encodings is based on the size of the encoding. The encoding with smaller size is always chosen.

Note that the *sequential* and *totalizer* encodings have similar size complexities. However, it has been observed that for solving MaxSAT the totalizer encoding is, in general, more efficient than the sequential encoding [MML11a]. Therefore, the sequential encoding is not considered in our portfolio of cardinality encodings.

Table 3.2: Instances solved by linear search algorithms with different cardinality encodings

Benchmark	#I	*at-most-k* − Linear search algorithms				
		Sequential	Totalizer	Sorters	PB	Dynamic
bcp-fir	59	51	53	51	10	53
bcp-hipp	55	38	40	42	18	42
bcp-msp	64	26	26	26	12	26
bcp-mtg	40	40	40	40	26	40
bcp-syn	74	32	32	32	21	32
circuit	4	4	4	4	4	4
haplotype	6	0	5	5	0	5
pbo-mqc	168	152	151	155	168	168
pbo-routing	15	15	15	15	13	15
protein	12	2	2	2	1	2
Total	497	360	368	372	273	387

3.4 Evaluation of the Encodings of Cardinality Constraints

This section evaluates the different cardinality encodings for MaxSAT algorithms, namely the linear search algorithm and the unsatisfiability-based algorithm described in the previous section for partial MaxSAT.

All experiments were run on the partial MaxSAT instances from the industrial category of the MaxSAT Evaluation 2011[1], which correspond to a set of 497 instances. The evaluation was performed on a computer with two AMD Opteron 6172 processors (2.1 GHz with 64 GB of RAM) running Fedora Core 13 with a timeout of 1,800 seconds (timeout used in the MaxSAT evaluations).

The different cardinality encodings were implemented on top of WBO (version 1.4, 2010) [MMSP09, MML10a]. The original WBO (version 1.0, 2009) [MMSP09] is an unsatisfiability-based solver that implements Algorithm 4 described in section 3.2.2. However, in WBO (version 1.4, 2010) [MML10a] the search can also be done by a linear search algorithm like the one described in section 3.2.1. In section 3.2, we have seen that the cardinality constraint *at-most-k* is used in linear search algorithms, whereas the cardinality constraint *at-most-one* is used in unsatisfiability-based algorithms. To evaluate these two types of cardinality constraints, we have run WBO using only one of the algorithms.

3.4.1 Encodings in Linear Search Algorithms

Table 3.2 shows the number of instances solved by our linear search algorithm using different cardinality encodings, including the dynamic encoding heuristic. Additionally to the cardinality

[1] `http://www.maxsat.udl.cat/11/`

Table 3.3: Number of times each encoding was selected by the dynamic encoding heuristic

#Solved	Dynamic Heuristic		
	Totalizer	Sorter	PB
387	100	247	40

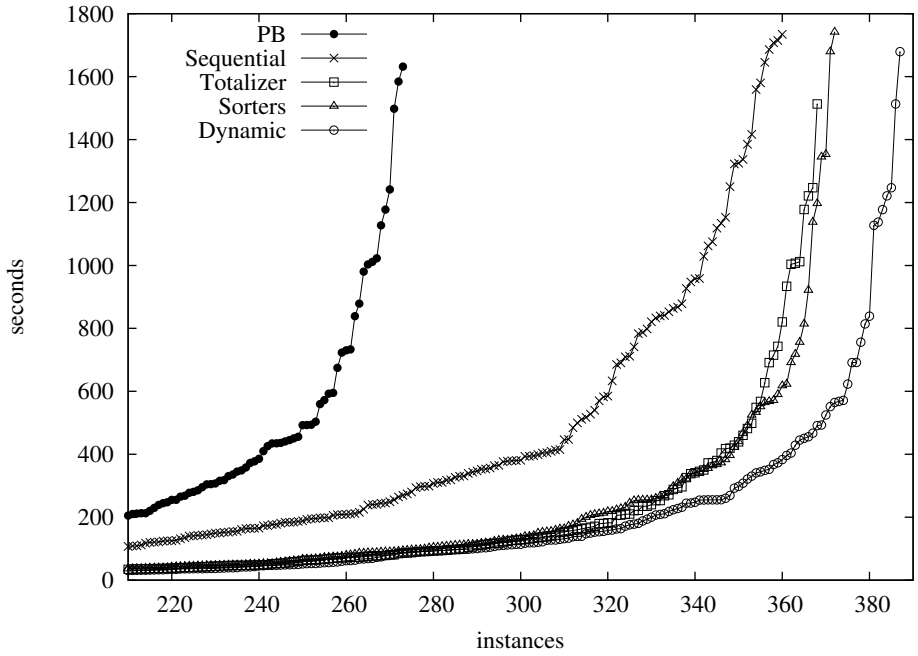


Figure 3.1: Cactus plot with run times of linear search solvers

encodings we also consider the pseudo-Boolean representation of cardinality constraints (PB), i.e. without encoding them into CNF.

For the *at-most-k* cardinality encodings, the *sorters* encoding performed better overall. As seen in the previous section, linear search algorithms use *incremental strengthening*, i.e. the cardinality constraint is only encoded once, when the first solution is found. Hence, the size of the cardinality constraint depends on the number of variables and on the upper bound value. Encoding cardinality constraints into CNF is therefore more effective when the number of variables is high (thousands) and the upper bound value is small when compared with the number of variables. This occurs, for example, in the `bcp-fir` benchmark. On the other hand, when given a cardinality constraint of size n with upper bound value close to $n/2$, using a *pseudo-Boolean* representation can be more effective than encoding the cardinality constraint into CNF. This is the case of the `pbo-mqc` benchmark set.

When considering the dynamic encoding heuristic, we can see that this heuristic outperforms

Table 3.4: Instances solved by unsatisfiability-based algorithms with different cardinality encodings

Benchmark	#I	*at-most-one* − Unsatisfiability-based algorithms								
		Pair.	Ladder	Bitwise	Comm.	Prod.	Seq.	Tot.	Sorters	PB
bcp-fir	59	44	50	46	52	47	49	50	49	44
bcp-hipp	55	21	22	21	21	22	21	23	20	20
bcp-msp	64	3	3	3	4	4	4	5	5	4
bcp-mtg	40	17	19	16	18	17	17	18	17	17
bcp-syn	74	34	35	35	35	35	34	34	34	34
circuit	4	0	1	1	1	1	1	1	1	0
haplotype	6	5	5	5	5	5	5	5	5	5
pbo-mqc	168	46	44	35	37	36	38	39	36	47
pbo-routing	15	15	15	15	15	15	15	15	15	15
protein	12	1	1	1	1	1	1	1	1	1
Total	497	186	195	178	189	183	185	191	183	187

all other cardinality encodings. This shows the importance of selecting the most adequate encoding for each problem instance.

Table 3.3 shows the number of times that each encoding was selected by the dynamic encoding heuristic when it was able to solve an instance. As expected, the *pseudo-Boolean* representation was the least used. Indeed, it was only used in the `pbo-mqc` benchmark set. The *totalizer* encoding was the second most used, and its application was mostly in the `bcp-fir` and `bcp-syn` benchmark sets. Overall, the *sorters* encoding was the most used by our dynamic encoding heuristic, since it usually has a smaller size than the *totalizer* encoding.

Figure 3.1 shows a cactus plot with the run times of linear search solvers. The cactus plot shows the sorted run times for each solver. Each point in the plot corresponds to a problem instance, where the y-axis corresponds to the wall clock time required by the solver and the x--axis corresponds to the accumulative number of instances solved until that time. The solvers considered were the solvers with the different cardinality encodings presented in Table 3.2. The *pseudo-Boolean* representation is shown to be much less effective than encoding the cardinality constraint into CNF. However, even between the different encodings we can see different run times. The *sequential* encoding is much less efficient than the *totalizer* and *sorter* encodings. Even though the *sorters* encoding is faster than the *totalizer* encoding, the performance of both encodings is comparable. Nevertheless, the dynamic encoding heuristic clearly outperforms all other encodings.

3.4.2 Encodings in Unsatisfiability-based Algorithms

Table 3.4 shows the number of instances solved by unsatisfiability-based algorithms when using different cardinality encodings. Additionally, we also consider the native representation of cardinality constraints given by the *pseudo-Boolean* representation. The first column of Table 3.4 shows the set of benchmarks. The second column shows the number of instances per benchmark set. The remaining columns show the number of instances solved when using the different *at-most-one* encodings.

For the *at-most-one* cardinality encodings, the *ladder* encoding performed best overall. However, for most benchmarks the number of solved instances by each encoding is different. If we would consider the best encoding for each instance, then it would be possible to solve 219 instances (more 24 instances than the ladder encoding). This shows that cardinality encodings can diversify the search, since each encoding enables solving different instances.

When the number of variables in the *at-most-one* constraint is small (less than a few hundred), then it is better to use the *pairwise* encoding or a *pseudo-Boolean* representation. This occurs, for example, in the `pbo-mqc` benchmark set. For these benchmark instances, the optimal value is usually low (around 10) and the average size of each unsatisfiable core is small (a few hundred clauses). Recall that every time an unsatisfiable core is found, a new *at-most-one* constraint is added to the formula. In partial MaxSAT, the number of unsatisfiable cores will be the same as the optimal value. Hence, if the optimal value is small, then the number of iterations will also be small. Moreover, the number of variables in the *at-most-one* constraint is the same as the size of the unsatisfiable core. As a result, for small unsatisfiable cores the number of variables in the *at-most-one* constraints will also be small.

With the exception of the *ladder* encoding, encodings that use auxiliary variables do not perform well on the `pbo-mqc` benchmarks. It has been observed that changing the branching heuristic not to branch on auxiliary variables may lead to better results [MSL07]. On the other hand, if the number of variables in the *at-most-one* constraint is large (several thousands), then it is better to encode the constraint into CNF. This can be observed in the `bcp-fir` benchmark instances where the unsatisfiable cores found by our algorithm are usually larger. Therefore, it is necessary to encode larger cardinality constraints (with thousands of variables).

3.5 Summary

This chapter described MaxSAT and its variants. MaxSAT algorithms that will be used in the remainder of the dissertation were presented, namely linear search algorithms and unsatisfiability-based algorithms. Even though cardinality constraints do not occur in MaxSAT formulations, they are often used by several MaxSAT algorithms. If one wants to continue using a SAT solver, then cardinality constraints must be encoded into CNF.

This chapter examined a large number of cardinality encodings and evaluated their performance for solving the MaxSAT problem. Linear search algorithms use *at-most-k* cardinality constraints. For the *at-most-k* cardinality constraint, the sorters encoding showed the best performance. In general, it is better to translate the *at-most-k* cardinality constraint into CNF. However, in some cases, using the native pseudo-Boolean representation can be more effective. Therefore, a dynamic encoding heuristic that selects the most adequate encoding for each cardinality constraint is proposed in this chapter. Unsatisfiability-based algorithms use *at-most-one* cardinality constraints. Overall, the ladder encoding showed the best performance for the *at-most-one* cardinality constraints. As expected, when the number of variables is small it is better to use the pairwise encoding or a pseudo-Boolean representation. On the other hand, when the number of variables in the cardinality constraint is large, it is better to encode the *at-most-one* cardinality constraint into CNF.

Linear search algorithms perform an upper bound search, whereas unsatisfiability-based algorithms perform a lower bound search. The orthogonality of these algorithms and the diversity of cardinality encodings are explored in the next chapter to build a parallel MaxSAT solver.

4

Parallel MaxSAT

Nowadays, extra computing power is not coming anymore from higher processor frequencies but rather from a growing number of cores and processors. In the last years, parallel SAT solvers have successfully exploited this new architecture. When compared with SAT instances, MaxSAT instances tend to be more intricate [Pap94]. When solving a MaxSAT instance, it is not sufficient to find an assignment that satisfies all clauses, but rather an assignment that satisfies all hard clauses and minimizes the sum of the weights of unsatisfied soft clauses. Hence, it comes as a natural step to develop parallel algorithms to MaxSAT, following the recent success in the SAT field.

This chapter presents PWBO, the first parallel solver for partial MaxSAT, and is organized as follows. First, approaches for parallel MaxSAT are presented, namely searching on the lower and upper bound values, portfolio-based approaches and search space splitting approaches. Next, we describe how learned clauses can be exchanged between the different parallel algorithms for MaxSAT. Finally, experimental results are presented that show the effectiveness of the proposed parallel MaxSAT solver.

4.1 Searching on the Upper and Lower Bound Values

Our parallel search is based on two orthogonal algorithms: (i) unsatisfiability-based algorithms that search on the lower bound of the optimal solution, i.e that perform *lower bound search*, and (ii) linear search algorithms that search on the upper bound of the optimal solution, i.e. that perform *upper bound search*. Therefore, we propose to perform a parallel search on both the upper

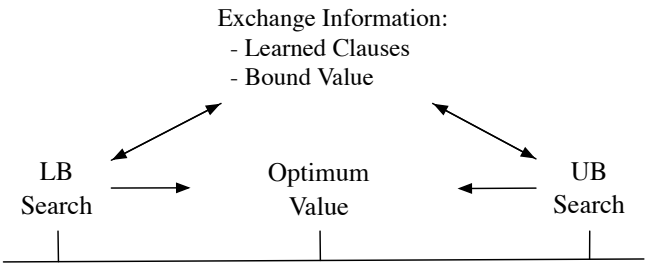


Figure 4.1: Solver architecture for two threads

bound and the lower bound of the optimal solution.

The parallel search for two threads can be seen as a generalization of WBO (version 1.4, 2010) [MML10a]. The original WBO (version 1.0, 2009) [MMSP09] is an unsatisfiability-based solver that implements Algorithm 4 described in section 3.2.2. However, in WBO (version 1.4, 2010) [MML10a] the search is initially performed by a linear search algorithm for 10% of the time limit given to solve the instance. If no solution has been found within this time, then it changes the search procedure to an unsatisfiability-based algorithm. When using two threads, our proposed parallel solver consists in using one thread to perform lower bound search by using an unsatisfiability-based algorithm and another thread to perform upper bound search by using a linear search algorithm. Figure 4.1 shows the architecture for two threads. A parallel search with these two orthogonal strategies results in a performance as good as the best strategy for each problem instance. However, if both threads cooperate through clause sharing, it is possible to perform better than the best strategy. Additionally, both strategies can also cooperate in finding the optimum value. If during the search the lower bound value provided by the unsatisfiability-based algorithm and the upper bound value provided by the other thread become the same, it means that an optimal solution has been found. As a result, it is not necessary for any of the threads to continue the search to prove optimality since their combined information already proves it.

Example 4.1. *Consider the following scenario. A partial MaxSAT formula φ is currently being solved by 2 threads. Thread t_1 is searching on the lower bound value of the optimal solution, and thread t_2 is searching on the upper bound value of the optimal solution. Suppose the initial lower and upper bound values are 0 and $+\infty$, respectively. Thread t_1 uses an unsatisfiability-based algorithm and iteratively increases the lower bound until the optimal value is found. On the other hand, thread t_2 uses a linear search algorithm and iteratively decreases the upper bound until the optimal value is found.*

Table 4.1: Configuration of PWBO with 2 threads

	PWBO-T2-PB		PWBO-T2	
	Encoding	Search	Encoding	Search
Thread t_1	PB	LB	Ladder	LB
Thread t_2	PB	UB	Dynamic	UB

Suppose that thread t_1 returns unsatisfiable for lower bound value 0, meaning that there is no solution that satisfies all soft clauses, i.e. that unsatisfies 0 soft clauses. If this is the case, the lower bound value is increased to 1. In the meantime, suppose that thread t_2 returns satisfiable with a model that unsatisfies 7 soft clauses of the original partial MaxSAT formula. If this is the case, the upper bound value is decreased to 7.

After a few iterations, consider that thread t_1 is currently searching with lower bound value 3, and that thread t_2 returns satisfiable with a model that unsatisfies 3 soft clauses of the original partial MaxSAT formula. If this is the case, then we can decrease the upper bound value to 3. Since the lower bound value is the same as the upper bound value, the optimal solution has been found by the combined information of both threads.

Our parallel partial MaxSAT solver is denoted by PWBO [MML11b, MML11a, MML12e], and it is implemented on top of WBO (version 1.4, 2010) [MMSP09, MML10a]. Table 4.1 shows the configuration of PWBO with 2 threads that will be evaluated in this chapter. For each thread (t_1, t_2) is given the encoding for the cardinality constraint (Encoding) and the kind of search performed by each thread (LB or UB). LB corresponds to the lower bound search performed by the unsatisfiability-based algorithm (Algorithm 3, section 3.2.2) and UB corresponds to the upper bound search performed by the linear search algorithm (Algorithm 2, section 3.2.1).

PWBO-T2-PB uses the native pseudo-Boolean representation for the lower and upper bound search. This version corresponds to the first version of PWBO with 2 threads [MML11b]. Moreover, PWBO-T2-PB can be seen as a direct generalization of WBO, since WBO also uses the native pseudo-Boolean representation for both lower and upper bound search.

PWBO-T2 corresponds to the version of PWBO run with 2 threads and that uses the *ladder* encoding [GN04, AM04] for the lower bound search and the dynamic encoding heuristic for the upper bound search. As seen in section 3.3, these encodings were the ones that performed the best for the lower and upper bound search, respectively.

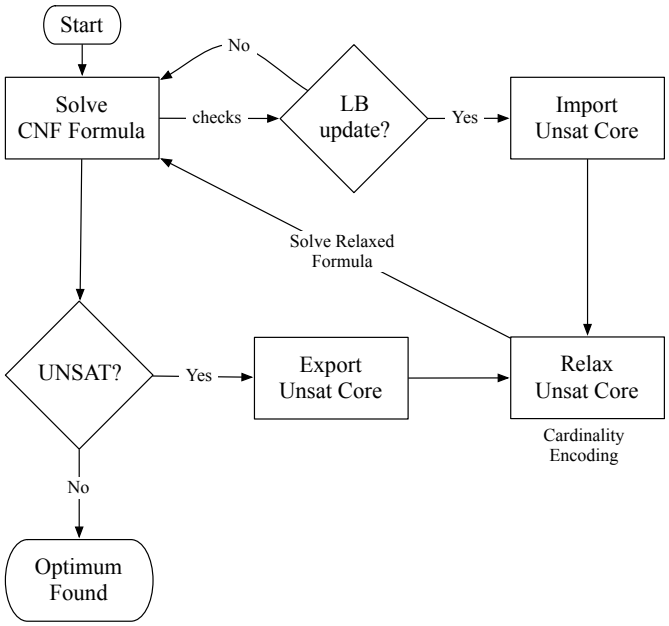


Figure 4.2: Parallel unsatisfiability-based algorithms

4.2 Portfolio Approaches

The previous section presented a parallel search solver for MaxSAT based on two orthogonal strategies. In the proposed approach, one thread is used for each strategy. For computer architectures with more than two cores, we can extend the previous idea by using several threads in the upper and lower bound search. However, if the algorithms that perform lower bound search are the same and the algorithms that perform upper bound search are also the same, then the gain from increasing the number of threads will be very small since all threads will be searching in a similar way. Therefore, it is important to increase the diversification of the search such that the search space is explored differently by each algorithm performing lower bound search and by each algorithm performing upper bound search. One solution is to exploit the variety of cardinality encodings by using a portfolio of algorithms using different encodings.

The portfolio approaches for parallel MaxSAT solving presented in this section are closely related to the portfolio approaches for parallel SAT solving presented in section 2.3. The main differences between these two approaches are: (i) our parallel portfolio MaxSAT solver uses two orthogonal algorithms, whereas parallel SAT solvers are usually based in the same algorithm; (ii) the diversification of the search is given by the different cardinality encodings, whereas parallel SAT solvers diversify the search through different heuristics.

Figure 4.2 illustrates parallel unsatisfiability-based algorithms. These algorithms work by iteratively identifying unsatisfiable cores and are based on Algorithm 3 presented in section 3.2. While solving the formula, the parallel algorithm checks if another thread has found a better lower bound value, i.e. if it has found an unsatisfiable core. If this is the case, then it imports the unsatisfiable core and relaxes the unsatisfiable core by adding relaxation variables to the soft clauses as described in Algorithm 3 in section 3.2. Similarly to Algorithm 3, for each soft clause in the identified unsatisfiable core, a new relaxation variable is added such that when this variable is assigned value 1, the soft clause becomes satisfiable. Moreover, an *at-most-one* cardinality constraint is also added to the relaxed formula such that only one of the newly created relaxation variables can be assigned value 1. Next, the solver checks if the formula remains unsatisfiable.

If a thread is not aware of a better lower bound value, then it continues the search process until it finds an unsatisfiable core or a solution to the formula. If it finds an unsatisfiable core, then it shares this unsatisfiable core with the remaining lower bound threads. Next, it relaxes the unsatisfied core as previously described and continues the search on the new formula. The procedure ends when the working formula becomes satisfiable and the solver returns an optimal solution.

To increase the diversification of the search, unsatisfiability-based algorithms use different cardinality encodings in the relaxation step. Any of the *at-most-one* encodings presented in section 3.3 can be used.

Note that there are a few details not shown in Figure 4.2. In particular, if the hard clauses are unsatisfiable then the MaxSAT instance is unsatisfiable and the solver terminates. Moreover, only one thread exports an unsatisfied core for each lower bound value. Before exporting an unsatisfiable core, the respective thread checks if its lower bound value is the greatest lower bound value among all threads. If this is the case, then it is safe to export the unsatisfiable core to the remaining threads. Otherwise, it discards its own unsatisfiable core and imports the unsatisfiable core that corresponds to the current lower bound value. Moreover, when a thread relaxes an unsatisfiable core, it updates its lower bound value.

Figure 4.3 illustrates parallel linear search algorithms. These algorithms are based on Algorithm 2 presented in section 3.2. Recall that the original MaxSAT formula φ is modified by adding a new relaxation variable r to each soft clause ω from φ, resulting in an equivalent formula φ_{UB} where one wants to minimize the number of relaxation variables assigned value 1. In the parallel algorithm, whenever a new solution is found for φ_{UB}, the upper bound value is updated and a new

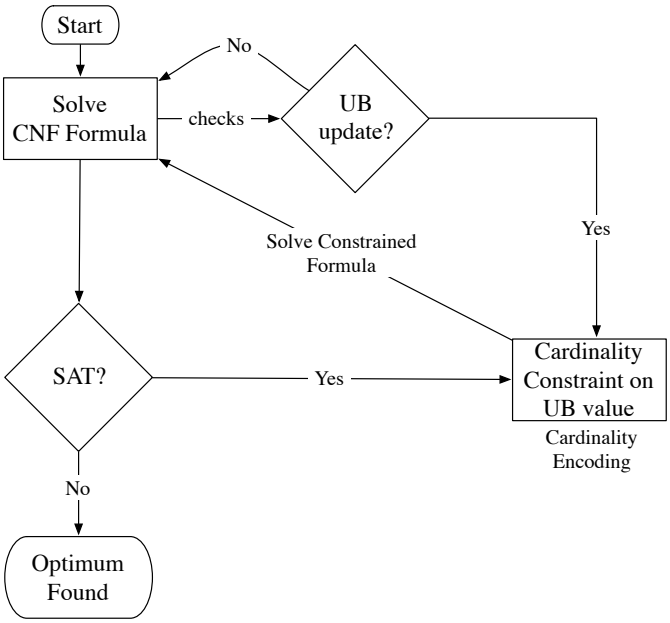


Figure 4.3: Parallel linear search algorithms

cardinality constraint on the relaxation variables is added such that all solutions with a greater or equal value are excluded. During search, each algorithm checks if there is a better upper bound value. If this is the case, it adds a cardinality constraint considering the new upper bound value. Afterwards, it restarts the search on the constrained formula.

The encodings used in the linear search algorithms support *incremental strengthening*. Since the upper bound value is always decreasing, the cardinality constraint only needs to be encoded when the first upper bound value is found. In the following iterations, one can assign truth value 0 to some specific literals in the encoding such that it restricts the cardinality constraint to the new upper bound value. Hence, all learned clauses from previous iterations remain valid and can therefore be kept.

To increase the diversification of the search, the parallel linear search algorithms should differ between themselves on the cardinality encoding that is used whenever a new cardinality constraint is added to the working formula. Any of the *at-most-k* encodings presented in section 3.3 can be used.

Example 4.2. *Consider the following scenario. A partial MaxSAT formula φ is currently being solved by 4 threads. Thread t_1 and thread t_2 are searching on the lower bound value of the optimal solution, and threads t_3 and t_4 are searching on the upper bound value of the optimal solution.*

Table 4.2: Configuration of PWBO-P with 4 and 8 threads

	PWBO-P-T4		PWBO-P-T8	
	Encoding	Search	Encoding	Search
Thread t_1	Commander	LB	Commander	LB
Thread t_2	Totalizer	LB	Totalizer	LB
Thread t_3	Sorters	UB	Ladder	LB
Thread t_4	PB	UB	Product	LB
Thread t_5	–	–	Sorters	UB
Thread t_6	–	–	PB	UB
Thread t_7	–	–	Sequential	UB
Thread t_8	–	–	Totalizer	UB

Suppose the initial lower and upper bound values are 0 and $+\infty$, respectively.

Suppose that thread t_1 returns an unsatisfiable core φ_C for the lower bound value 0, meaning that there is no solution that satisfies all soft clauses. Thread t_2 periodically checks (every 300 conflicts) if another thread has found an unsatisfiable core for its current lower bound value. Since thread t_1 found an unsatisfiable core, then thread t_2 stops its search and imports φ_C. Thread t_2 relaxes φ_C as described in Algorithm 3, updates its current lower bound value to 1, and continues its search on the new lower bound value.

In the meantime, consider that thread t_3 returns satisfiable with a model that unsatisfies 7 soft clauses of the original partial MaxSAT formula. If this is the case, then the upper bound value is decreased to 7. Thread t_4 periodically checks if another thread has found a smaller upper bound value. Since thread t_3 found an upper bound value of 7, then t_4 stops its search and updates its current upper bound value to 7.

This procedure is repeated until the parallel unsatisfiability-based algorithms or the parallel linear search algorithms find an optimal solution or until the lower bound value is the same as the upper bound value.

Table 4.2 shows the configuration of PWBO-P (where P stands for portfolio) with 4 and 8 threads that will be used in the remainder of the dissertation. To maintain a balance between lower and upper bound search, PWBO-P always uses the same number of threads for the upper bound and the lower bound search. To build a portfolio of encodings for 4 and 8 threads we took into consideration the evaluation of the encodings of cardinality constraints presented in section 3.3. Note that it may occur the situation where a cardinality encoding has an overall poor performance, but is the only one to be able to solve a given set of instances. In this case, it is interesting to incorporate such an encoding into a portfolio approach.

With 4 threads, PWBO-P-T4 uses the *commander* and *totalizer* encodings for the lower bound search and *sorters* and *pseudo-Boolean* encodings for the upper bound search. Although the *ladder* encoding performed better for the *at-most-one* constraint, it was mainly on solving the `bcp-mtg` and `pbo-mqc` benchmark sets. However, the performance of the upper bound search procedure on those instances is much better than the performance of the lower bound search. Therefore, the main gains of the *ladder* encoding are already covered by the *at-most-k* encodings for the upper bound search. A similar reasoning is applied to the *pseudo-Boolean* representation for the upper bound search. Even though this representation is less effective in general, it is the best performing encoding for solving the `pbo-mqc` benchmark set. Hence, this portfolio of cardinality encodings allows for a diversification of the search in all benchmarks.

With 8 threads, PWBO-P-T8 can use more encodings and therefore can further increase the diversification of the search. For the upper bound search, all four available encodings are used. For the lower bound search, we have selected the following encodings: *commander*, *totalizer*, *ladder* and *product*. The *ladder* encoding was now selected due to its overall robustness. On the other hand, even though the product encoding is less effective than other encodings, we have noticed that when it solves a given instance, it can be faster than other encodings. This has already been observed before [FG10]. Hence, for speedup reasons, we have decided to include the product encoding on our portfolio of cardinality encodings with 8 threads.

4.3 Search Space Splitting Approaches

Another approach that can be done for computer architectures with more than two cores is to split the search space by searching on different local upper bound values. In this parallel search, if n cores are available, then one thread is used to search on the lower bound, another thread is used to search on the upper bound, and the remaining $n - 2$ threads will search on different local upper bound values. The local upper bound values restrict the search space by enforcing a fixed upper bound value of the optimal solution. Since this fixed upper bound value is restricted to each thread, it is named as *local upper bound value*. The search performed by these threads is named as *local upper bound search*. The iterative search on different local upper bound values leads to constant updates on the lower and upper bound values that reduce the search space. Next, an example of this approach is described. Afterwards, we present a more detailed description of the local upper bound search.

Example 4.3. *Consider a partial MaxSAT formula φ as input. For the input formula, one can easily find initial lower and upper bounds. Suppose the initial lower and upper bound values are 0 and $+\infty$, respectively. Moreover, consider also that the optimal value is 3 and our goal is to find it using four threads: t_1, t_2, t_3 and t_4. Thread t_1 applies an unsatisfiability-based algorithm (i.e., searches on the lower bound of the optimal solution). This thread starts with a lower bound of 0 and will iteratively increase the lower bound until the optimal value is found.*

Thread t_2 searches on the upper bound value of the optimal solution. Hence, thread t_2 starts its search with upper bound value of $+\infty$. Threads t_3 and t_4 search on different local upper bound values. For example, if we divide the interval of possible local upper bound values, then threads t_3 and t_4 can start their search with local upper bound values of 3 and 7, respectively.

Suppose that thread t_3 finishes its computation and finds that the formula is unsatisfiable for an upper bound value of 3. This means that there is no solution with values 0, 1 and 2. Therefore, the lower bound value can be updated to 3. Thread t_3 is now free to search on a new local upper bound value, for example 5. In the meantime, thread t_4 finds a solution with value 6. Hence, the upper bound value can be updated to 6. Thread t_2 updates its upper bound value to 6 and thread t_4 is now free to search on a different local upper bound value, for example 4. Afterwards, consider that thread t_2 finds a solution with value 3. Again, the upper bound value can be updated to 3. Since the lower bound value is the same as the upper bound value, the optimal value has been found and the search terminates.

The splitting approaches for parallel MaxSAT solving presented in this section can be related to the search space splitting approaches for parallel SAT solving presented in section 2.3. In parallel SAT solving, search space splitting consists of dividing the search space into disjoint subspaces that can be explored in parallel. On the other hand, the proposed splitting approach for parallel MaxSAT uses the possible range of values of the optimal solution to split the search space. This approach splits the search space into different subspaces to be searched independently by each thread. Even though the union of the subspaces covers the entire search space, they are not necessarily disjoint.

Splitting approaches for parallel MaxSAT incorporate three types of algorithms: unsatisfiability-based, linear search and local linear search. The unsatisfiability-based and linear search algorithms used by the threads that are searching in the lower and upper bound values of the optimal solution are the same as the ones presented in Figures 4.2 and 4.3 in the previous section. In what follows we will describe the algorithm for parallel local linear search algorithms that is used by

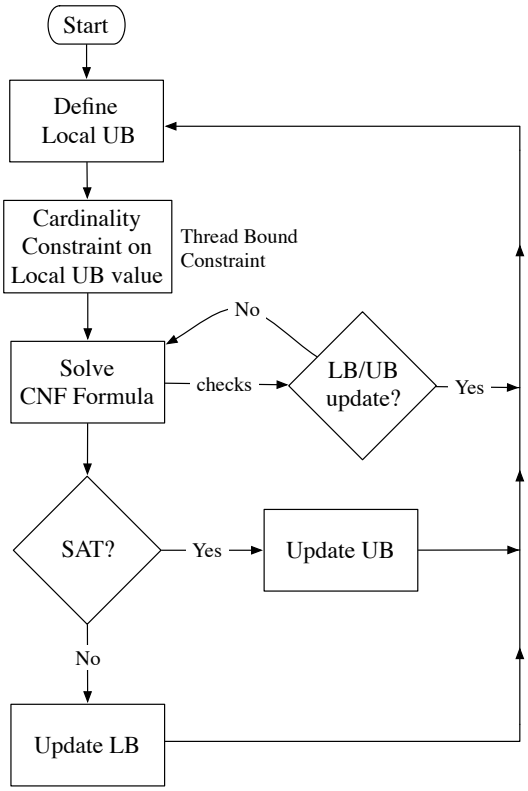


Figure 4.4: Parallel local linear search algorithms

the remaining threads to perform local upper bound search.

Figure 4.4 illustrates parallel local linear search algorithms. Similarly to the linear search algorithms, the original MaxSAT formula φ is modified by adding a new relaxation variable r to each soft clause ω from φ, resulting in an equivalent formulation φ_{UB} where the goal is to minimize the number of relaxation variables assigned value 1.

These algorithms start by defining their local upper bound. Initially, we set the lower bound value to 0 and the upper bound value to $+\infty$. In practice, the upper bound value does not need to be set to $+\infty$ since it suffices to set it to the number of soft clauses in φ plus 1. Consider a MaxSAT formula φ with s soft clauses and k local threads, $t_1, \ldots, t_k$. The initial local upper bound value b_j of a thread t_j is given by $j \times \lfloor (s+1)/(k+1) \rfloor$.

Next, thread t_j adds a cardinality constraint of the form $\sum r_i \leq b_j - 1$ to exclude solutions with a value greater than or equal to b_j. Let this cardinality constraint be labeled the *thread bound constraint*. If a cardinality encoding is used, then all clauses that were created to encode the cardinality constraint will be labeled as thread bound constraints.

After adding a thread bound constraint, the algorithm starts the search. During the search, the algorithm checks if another thread has found a lower bound value that is greater than the thread current local upper bound value or an upper bound value that is smaller than the thread current local upper bound value. If one of these cases occurs, then the algorithm will terminate its search and a new local upper bound value is defined. Next, the search restarts using the new upper local value.

If the algorithm is not aware of a better lower or upper bound value, then it continues the search process until it finds a solution or proves that no solution exists for the current local upper bound value. If a solution is found, then the algorithm updates the upper bound value. Otherwise, if it proves that no solution exits, then the lower bound value is updated. In both cases, a new local upper bound value is set and the search restarts.

Currently, parallel local linear search algorithms are not computing an unsatisfiable core when a new lower bound value is found. Therefore, no unsatisfiable cores are exported to the unsatisfiability-based algorithm that is searching on the lower bound value. Moreover, the thread searching on the lower bound does not update its lower bound value to the new lower bound value found by the parallel local linear search algorithms.

There are a few details not shown in figure 4.4. Updates to the lower and upper bound values only take place when the new values improve the current ones. Additionally, when a thread is assigned a new local upper bound value after finding a solution or proving that a solution does not exist, this new local upper bound value covers the broadest range of yet untested bounds. More formally, the new local upper bounds are chosen as follows. Let $B = \langle b_1, b_2, \ldots, b_{k-1}, b_k \rangle$ be a sorted list where b_1 corresponds to the lower bound value and b_k corresponds to the upper bound value, while the remaining b_j are the non-aborted thread local upper bound values. Let $[b_{m-1}, b_m]$, where $1 < m \leq k$, define an interval such that for all $1 < j \leq k$ we have $b_m - b_{m-1} \geq b_j - b_{j-1}$. In this case, the new upper bound value of the aborted thread is $\lfloor (b_m + b_{m-1})/2 \rfloor$. The sorted list B is updated with the new value and this process is repeated for each aborted thread.

Example 4.4. *Consider the following scenario. A partial MaxSAT formula φ is currently being solved by 4 threads. Thread t_1 is searching on the lower bound value of the optimal solution, and thread t_2 is searching on the upper bound value of the optimal solution. The current lower and upper bound values are 5 and 10, respectively. Thread t_3 is searching on a local upper bound with value 8 and thread t_4 is computing a new local upper bound value. The sorted list B corresponds to $B = \langle 5, 8, 10 \rangle$. Thread t_4 will now determine the largest interval between two consecutive values*

in B, i.e. $[5, 8]$. *Therefore, the new upper local upper bound value of thread* t_4 *will be given by* $\lfloor (8 + 5)/2 \rfloor = 6$.

4.4 Clause Sharing

Conflict-driven clause learning [MSS96, ZMMM01] is crucial for the efficiency of modern SAT solvers. After detecting a conflict, i.e. a sequence of assignments that make a clause unsatisfiable, a new clause is learned to prevent the same conflict from occurring again in the subsequent search. The new clause results from the analysis of the implication graph which represents the dependencies between assignments. A more detailed explanation can be found in the literature [MSS96, ZMMM01].

Clause learning is also essential to the efficiency of many modern MaxSAT solvers. In the context of parallel solving, sharing learned clauses is expected to help to further prune the search space and boost the performance of a parallel solver. Similarly to parallel SAT solving [HJS09b], only learned clauses that have less than a given number of literals are shared among all threads.

In our parallel solver, we start by sharing learned clauses that have 8 or fewer literals. However, in our parallel solver not all learned clauses can be shared among all threads. This is due to the fact that the working formulas are different. As previously explained, unsatisfiability-based algorithms work directly with the input formula φ, while algorithms that perform a linear search on the upper bound value add relaxation variables to the soft clauses, resulting in formula φ_{UB}. In order to define the conditions for safe clause sharing, we start by defining soft and hard learned clauses.

Definition 4.1 (Soft and Hard Learned Clauses)**.** *If the conflict analysis procedure used in the unsatisfiability-based algorithm involves at least one soft clause used in the implication graph, then the generated learned clause is labeled as soft. On the other hand, if only hard clauses are used, then the generated learned clause is labeled as hard.*

Since φ contains both soft and hard clauses, both soft and hard learned clauses can be added to the formula. On the other hand, φ_{UB} only has hard clauses, and as a result can only add hard learned clauses. Nevertheless, as mentioned before, φ_{UB} contains additional relaxation variables that are not present in φ. When using cardinality encodings, we also have to take into account the auxiliary variables used by those encodings. Therefore, each thread may contain variables not present in the other threads. Moreover, threads that perform local upper bound search contain thread bound constraints. These constraints cannot be shared among all threads, since they

Table 4.3: Type of learned clauses created by the different algorithms

Learned Clause	Algorithms		
	LB	Local UB	UB
Soft	✓		
Hard	✓	✓	✓
Local		✓	
w/ encoding vars	✓	✓	✓

are only valid if the optimum value is smaller than the upper bound value of the thread. The same sharing rules must apply to conflict-driven learned clauses that depend on the thread bound constraint. Therefore, it is necessary to define what is a local constraint and in which conditions it can be shared with other threads.

Definition 4.2 (Local Constraint)**.** *The thread bound constraint is labeled as a local constraint. Let ω be a conflict-driven learned clause and let φ_ω be the set of constraints used in the implication graph to learn ω. The new clause ω is defined as a local constraint if at least one constraint in φ_ω is a local constraint.*

Table 4.3 summarizes the different kinds of learned clauses that can be created by the different algorithms. As mentioned before, algorithms that perform lower bound search can create soft, hard and learned clauses with auxiliary variables from the cardinality encodings. On the other hand, algorithms that perform local bound search can create hard, local and learned clauses with auxiliary variables. Finally, algorithms that perform upper bound search can create hard learned clauses and learned clauses with auxiliary variables. However, not all clauses that are created can be shared among the different algorithms. To be safe, the sharing procedure between the different algorithms is as follows:

- Hard learned clauses from unsatisfiability-based algorithms that do not have encoding variables can be safely shared with the other threads.

- Soft learned clauses from unsatisfiability-based algorithms are not shared with the other threads. These clauses may not be valid for formulas φ_{UB} and cannot be shared with the algorithms that perform linear search on the upper bound.

  Notice that these clauses could eventually be shared with other threads that are using unsatisfiability-based algorithms. However, it would be necessary to establish an equivalence between the relaxation variables of the learned soft clause and the relaxation variables

of the importing thread. Since variables are created for producing the encoding of cardinality constraints, the identification of the relaxation variables may differ between threads. Even though it would be possible to share soft learned clauses between unsatisfiability-based algorithms, this is currently not implemented in our parallel solver.

- Hard learned clauses generated when solving φ_{UB} can be shared with the other threads if the learned clause is not a local constraint and if it does not contain relaxation or auxiliary variables.

- Hard learned clauses that are local constraints generated when solving φ_{UB} cannot be safely shared with the lower bound threads. However, local constraints that do not contain auxiliary variables can be shared between upper bound threads. Sharing local constraints depends on the upper bound value of the thread. If an importing thread has an upper bound value smaller than or equal to the upper bound value of the exporting thread, then the import is safe. Otherwise, the import may be unsafe and the respective clauses are not shared.

Finally, between iterations of the unsatisfiability-based algorithms, the working formulas φ are also extended with additional relaxation variables. However, since these variables are added to soft clauses, if a conflict-based learned clause contains any relaxation variable, then it will necessarily be considered a soft clause. This is due to the fact that at least one soft clause would have been used in the learning procedure.

4.4.1 Integration of Learned Clauses

Whenever a learned clause is generated, if the clause size is smaller than the current cutoff and if the clause meets the safe sharing conditions, it is exported as a learned clause to the other threads. Later on, when a thread checks if there is a better lower or upper bound value, it also imports the learned clauses that were shared by other threads. Since importing clauses occurs during the search, the learned clauses have to be integrated in the context of the current search space. Hence, the addition of a shared clause ω has to take into consideration the following cases:

- ω is a unit clause: a restart is forced and the corresponding literal is assigned.

- ω is unit in the current context: the SAT algorithm backtracks to the highest decision level of the assigned variables in ω. After backtracking, the unassigned literal is assigned and propagated.

- ω is unsatisfied in the current context: the SAT algorithm backtracks to the highest decision level of the variables in ω. Conflict analysis is performed to allow further backtracking. Moreover, during the conflict analysis procedure a new clause is learned.

- ω is satisfied in the current context: if exactly one literal in ω is satisfied and the remaining literals are falsified, and if the decision level of the satisfied literal is higher than the decision levels of all falsified literals, then the algorithm backtracks to the highest decision level among the falsified literals.

In the remaining cases the learned clause is simply added to the importing thread and no backtracking is needed. The integration procedure must be applied in order to ensure the correctness of the solver. A similar procedure is done in the parallel SAT solver MANYSAT [HJS09b].

4.5 Experimental Results

This section evaluates the different versions of PWBO (version 1.0, 2011) [MML11b, MML11a, MML12e]. Using two threads, PWBO-T2 searches on the lower and upper bound values of the optimal solution. For more than two threads, PWBO-P and PWBO-S (where S stands for split) are evaluated. The additional threads in PWBO-P search on the lower and upper bound values of the optimal solution with different cardinality encodings for each thread. On the other hand, in PWBO-S the additional threads perform a parallel search on different upper bound values of the optimal solution.

All experiments were run on the partial MaxSAT instances from the industrial category of the MaxSAT Evaluation 2011[1]. The evaluation was performed on a computer with two AMD Opteron 6172 processors (2.1 GHz with 64 GB of RAM) running Fedora Core 13 with a timeout of 1,800 seconds (wall clock time). For the parallel solvers, results were obtained by running each solver ten times on each instance. Similarly to what is done when analyzing randomized solvers, the median time was taken into account. This means that an instance must be solved by at least five of the ten runs to be considered solved.

4.5.1 Multithread based on Lower and Upper Bound Search

Figure 4.5 shows a cactus plot with run times for the sequential solvers WBO and WBO-CNF and the different versions of PWBO-T2.

[1] `http://maxsat.ia.udl.cat/`

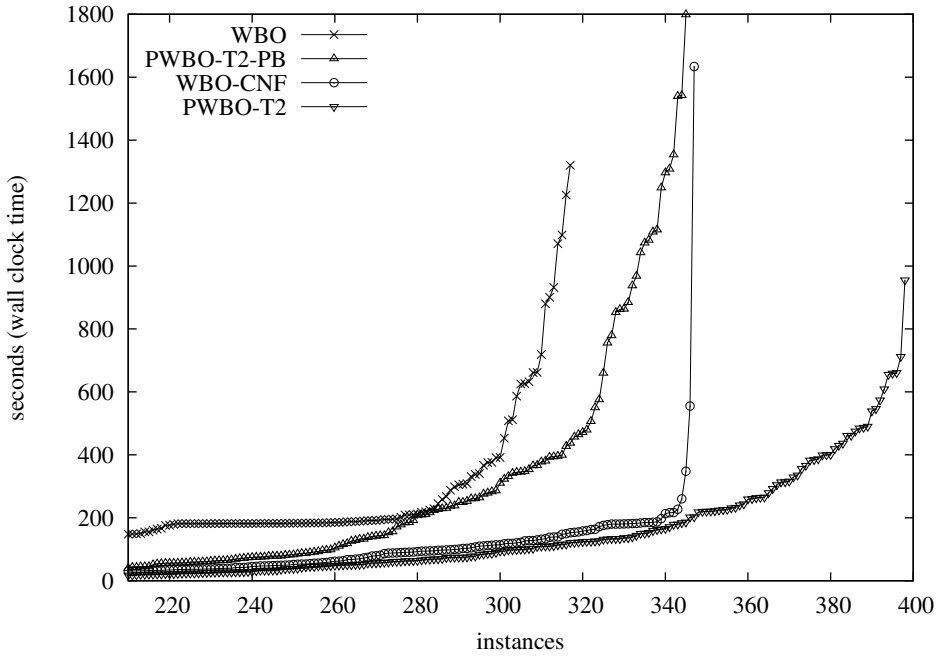


Figure 4.5: Cactus plot with run times of WBO, WBO-CNF and the different versions of PWBO-T2

WBO-CNF corresponds to the version of WBO using CNF encodings. WBO uses the *pseudo-Boolean* encoding for both the linear search algorithm and the unsatisfiability-based algorithm. On the other hand, WBO-CNF uses the dynamic encoding heuristic for the linear search algorithm and the *ladder* encoding for the unsatisfiability-based algorithm.

PWBO-T2-PB outperforms the sequential solver WBO. PWBO-T2-PB uses the *pseudo-Boolean* representation for both the linear search algorithm and the unsatisfiability-based algorithm. Therefore, this version of PWBO can be seen as a generalization of WBO for 2 threads.

Section 3.4 has shown that encoding cardinality constraints into CNF greatly improves the performance of the sequential solver. Due to the CNF encodings, WBO-CNF is able to outperform WBO and PWBO-T2-PB. On the other hand, PWBO-T2 clearly outperforms PWBO-T2-PB, as well as the sequential solvers WBO and WBO-CNF. PWBO-T2 uses the *ladder* encoding for the lower bound search and the dynamic encoding heuristic for the upper bound search. Therefore, this version of PWBO can be seen as a generalization of WBO-CNF for 2 threads.

Since we are using two threads, the solution can be found in three ways: by the lower bound search, by the upper bound search or by the cooperation between the lower bound search and the upper bound search. If the lower bound value is the same as the upper bound value, then the optimal solution has been found by the information of both searches. Table 4.4 shows the

Table 4.4: Number of instances solved by PWBO-T2 divided by lower bound search, upper bound search and cooperation between searches

	PWBO-T2			
Benchmark	#Solved	Lower	Upper	Coop.
bcp-fir	56	24	18	14
bcp-hipp	42	4	29	9
bcp-msp	26	0	23	3
bcp-mtg	40	0	40	0
bcp-syn	40	14	14	12
CircuitTrace	4	0	4	0
Haplotype	5	5	0	0
pbo-mqc	168	2	165	1
pbo-routing	15	11	2	2
PROTEIN_INS	2	0	2	0
Total	398	60	297	41

number of instances that were solved in each case. As expected, the upper bound search solves the largest number of instances of the two-threaded version solving 297 out of the 398 solved instances. However, the lower bound search contributes to the performance of PWBO-T2 by solving 60 instances. Moreover, 41 instances are solved by the combined information of the lower bound search and the upper bound search. For these instances the cooperation speeds up the solving process. Since the lower bound value was found to be the same as the upper bound value, it is not necessary for any of the threads to continue the search to prove optimality since their combined information already proves it. Additionally, Table 4.4 provides a strong stimulus to further improve the lower bound search, since a more efficient lower bound search may improve the overall performance of the parallel solver.

4.5.2 Multithread based on Portfolio

PWBO-P is based on a portfolio approach where each thread uses a different cardinality encoding [MML11a]. This section evaluates the performance of PWBO-P with 4 and 8 threads. As described previously, PWBO-P always uses the same number of threads for the upper bound and lower bound search. Section 4.2 presented the configuration used by PWBO-P with 4 (PWBO-P-T4) and 8 threads (PWBO-P-T8). With 4 threads, PWBO-P-T4 uses the *commander* and *totalizer* encodings for the lower bound search and the *sorters* and *pseudo-Boolean* encodings for the upper bound search. With 8 threads, PWBO-P-T8 can use more encodings and therefore can further increase the diversification of the search. For the upper bound search, all four available encodings are used. For lower bound search, PWBO-P-T8 uses the following encodings: *commander*, *totalizer*,

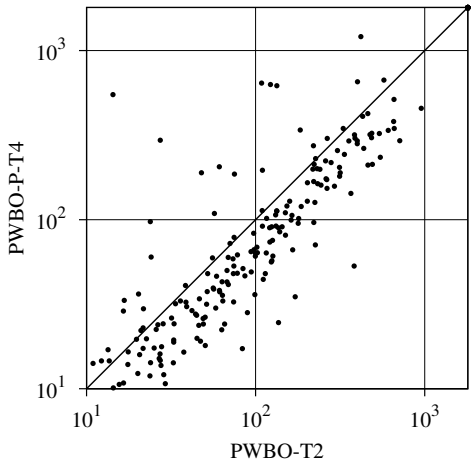


Figure 4.6: Comparison between run times of PWBO-T2 and PWBO-P-T4

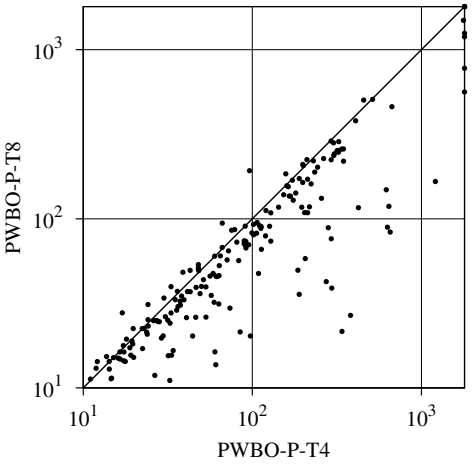


Figure 4.7: Comparison between run times of PWBO-P-T4 and PWBO-P-T8

ladder and *product*.

Figure 4.6 shows a scatter plot with run times of PWBO-T2 and PWBO-P-T4. Each point in the plot corresponds to a problem instance, where the x- axis corresponds to the run time required by PWBO-T2 and the y-axis corresponds to the run time required by PWBO-P-T4. Instances that are trivially solved by both approaches (in less than 10 seconds) are not shown in the plot. The portfolio approach with 4 threads outperforms PWBO-T2 on most instances. However, there are some instances where PWBO-T2 performs better. PWBO-T4 does not use the *totalizer* encoding whereas the PWBO-T2 uses a dynamic encoding heuristic that chooses that encoding for some instances. PWBO-T4 may be further improved if we consider a variation of the presented dynamic encoding heuristic. For the upper bound search, one thread could always use the *sorters* encoding whereas the other thread could use a dynamic encoding heuristic that would select between the *totalizer* encoding and the *pseudo-Boolean* representation.

Figure 4.7 compares PWBO-P-T4 with PWBO-P-T8. Notice that PWBO-P-T8 is able to solve more instances and with better run times than PWBO-P-T4 on most of the instances. This shows that even with 8 threads we are still able to increase the diversification of the search by adding different cardinality encodings.

4.5.3 Multithread based on Splitting

PWBO-S is based on splitting the search space according to what was described in section 4.3. One thread searches on the lower bound value of the optimal solution, another thread searches on the

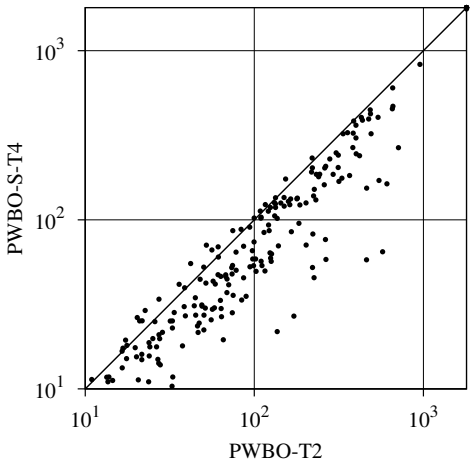


Figure 4.8: Comparison between run times of PWBO-T2 and PWBO-S-T4

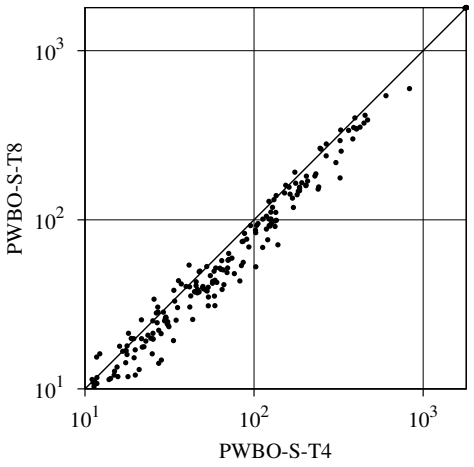


Figure 4.9: Comparison between run times of PWBO-S-T4 and PWBO-S-T8

upper bound value of the optimal solution, and the remaining threads search on different upper bound values [MML11b]. PWBO-S uses the dynamic encoding heuristic proposed in section 3.3 in all threads that are searching on the upper bound value of the optimal solution. As for the lower bound search, it uses the *ladder* encoding. For 4 and 8 threads we denote PWBO-S as PWBO-S-T4 and PWBO-S-T8, respectively.

Figure 4.8 shows a scatter plot comparing run times of PWBO-T2 and PWBO-S-T4. The plot clearly shows that PWBO-S-T4 outperforms PWBO-T2, thus showing that the performance of the solver clearly improves with the increase of the number of threads from 2 to 4.

Figure 4.9 shows a scatter plot comparing run times of PWBO-S-T4 and PWBO-S-T8. Even though there is a slight improvement in time with the increase of the number of threads from 4 to 8, it is not as clear as before. With the split strategy, using more threads increases the number of threads that are searching on local upper bound values of the optimal solution. If the interval between the lower and upper bound values is small, then the threads that are searching on local upper bound values may be searching on similar values, thus performing redundant search. This may explain why the performance with 8 threads is not significantly better than the performance with 4 threads.

4.5.4 Impact of Clause Sharing

In section 4.4 we have described the sharing mechanism of PWBO. Sharing learned clauses is expected to further prune the search space and boost the performance of the parallel solver.

Table 4.5: Speedup gain of sharing learned clauses

Solver	Speedup
PWBO-T2	1.26
PWBO-S-T4	1.28
PWBO-S-T8	1.34
PWBO-P-T4	1.36
PWBO-P-T8	1.37

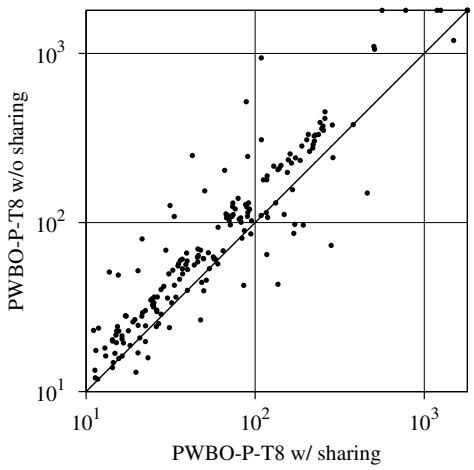


Figure 4.10: Comparison between run times of PWBO-P-T8 with and without clause sharing

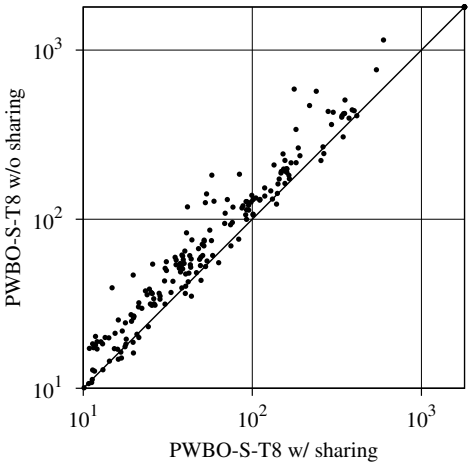


Figure 4.11: Comparison between run times of PWBO-S-T8 with and without clause sharing

To evaluate the impact of sharing learned clauses we run the different versions of PWBO with and without clause sharing. Table 4.5 shows the speedup gain of sharing learned clauses. The speedup is determined by the ratio between the total solving time of the solver with and without clause sharing. Only instances that were solved by the two versions, i.e. with and without clause sharing are considered for the total solving time. Therefore, the speedup shows how many times the solver with clause sharing was faster than the solver without clause sharing. For example, PWBO-T2 shows a speedup of 1.26, i.e. it was 1.26× faster when learned clauses were shared.

Sharing learned clauses has a clear speedup on the solving times of the solvers. Moreover, increasing the number of threads increases the gains of sharing learned clauses. The main improvement from clause sharing is in the speedup of the solver, since the number of solved instances does not increase significantly with clause sharing. For example, PWBO-S-T8 solves the same number of instances with and without clause sharing. The largest improvement can be seen in PWBO-P-T8 since it can solve more 4 instances with clause sharing.

Table 4.6: Number of instances solved by each solver and speedup of PWBO on the instances solved by all solvers

	#Solved	Speedup
WBO-CNF	347	1.00
PWBO-T2	398	1.36
PWBO-P-T4	399	1.67
PWBO-S-T4	399	2.19
PWBO-S-T8	399	2.36
PWBO-P-T8	403	2.57

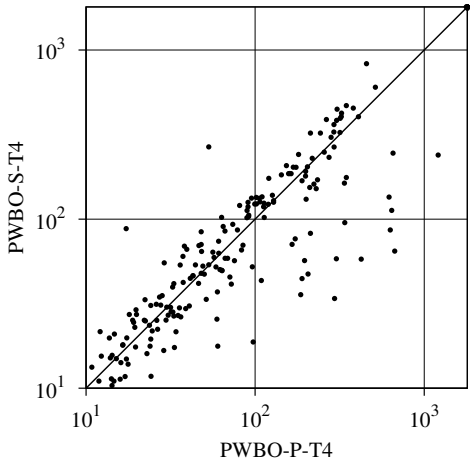


Figure 4.12: Comparison between run times of PWBO-P-T4 and PWBO-S-T4

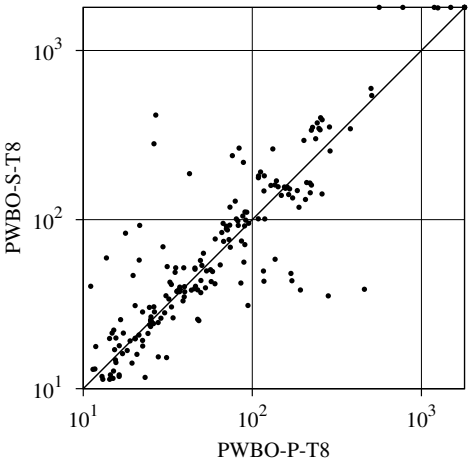


Figure 4.13: Comparison between run times of PWBO-P-T8 and PWBO-S-T8

A more detailed view of the impact of clause sharing can be seen in Figures 4.10 and 4.11. They provide scatter plots with the run times of PWBO-P-T8 and PWBO-S-T8 with and without sharing. It is clear that sharing learned clauses speedup the parallel solvers.

4.5.5 Comparison between Portfolio and Splitting

This section compares the different versions of PWBO. Table 4.6 shows the number of instances solved by each solver and the speedup of PWBO when compared to WBO-CNF on instances that were solved by all solvers. The results are clear: all parallel solvers outperform the sequential solver. Moreover, we can see a clear improvement between PWBO-T2 and the parallel solvers with 4 and 8 threads. The speedup increases with the number of threads being 1.4× faster with 2 threads, 2.2× faster with 4 threads, and 2.6× faster with 8 threads. PWBO-S-T4 solves the same number of instances as PWBO-P-T4 but has a larger speedup. However, when using 8 threads PWBO-P-T8 has a larger speedup than PWBO-S-T8. This supports our previous findings that PWBO-P scales

better than PWBO-S with an increasing number of threads. Indeed, PWBO-S-T8 is only 1.1× faster than PWBO-S-T4, whereas PWBO-P-T8 is 1.4× faster than PWBO-P-T4.

Figures 4.12 and 4.13 compare the portfolio approach with the splitting approach. Figure 4.12 compares PWBO-P and PWBO-S with 4 threads. PWBO-P-T4 slightly outperforms PWBO-S-T4 on the run times for most instances. However, there are several instances that PWBO-S-T4 is able to solve much faster than PWBO-P-T4. This explains why the overall speedup of PWBO-S-T4 is larger than PWBO-P-T4.

Figure 4.13 compares PWBO-P and PWBO-S with 8 threads. PWBO-P-T8 solves more 4 instances than PWBO-S-T4 and outperforms PWBO-S-T8 on the run times for most of the instances. Moreover, the number of instances for which PWBO-S-T8 clearly outperforms PWBO-P-T8 is small. This explains why the overall speedup of PWBO-P-T8 is larger than PWBO-S-T8.

4.6 Summary

This chapter introduced PWBO, the first parallel solver for partial MaxSAT. This work was in part motivated by the recent success of parallel SAT solvers and by the fact that parallel algorithms for Boolean optimization are just starting.

Three versions of PWBO were proposed. The first version, PWBO-T2, uses two threads, one thread searching on the lower bound value of the optimal solution, and another thread searching on the upper bound value of the optimal solution. The second version, PWBO-P, is based on a portfolio approach using several threads to simultaneously search on the lower and upper bound values of the optimal solution. These threads differ between themselves in the encoding used for cardinality constraints, thus increasing the diversification of the search. The third version, PWBO-S, is based on a splitting approach searching on different values of the upper bound. The parallel search on the local upper bound values leads to updates on the lower and upper bound values that will reduce the search space.

For two threads, experimental results show that most instances are solved by the upper bound search. Even though the lower bound search cooperates in solving the instances, it does not perform as well the upper bound search. As future work, we propose to improve the lower bound search algorithm, since a more efficient lower bound search may improve the overall performance of PWBO.

Experimental results also show that PWBO improves in performance with the increasing number of threads. With 4 threads, PWBO-S presented the best overall performance. However, the

performance of PWBO-S with 8 threads is only slightly better than with 4 threads. On the other hand, PWBO-P performance improves significantly when increasing the number of threads from 4 to 8. This shows that even with 8 threads, using different cardinality encodings still increases the diversity of the search.

This chapter also evaluated the impact of clause sharing in parallel MaxSAT solving. Experimental results with clause sharing show that sharing clauses does not have a strong impact on the number of solved instances. Nevertheless, the running times of PWBO are greatly improved when sharing learned clauses between threads.

The parallel approaches presented in this chapter were proposed for solving partial MaxSAT. However, they can also be used for solving weighted partial MaxSAT problems. Future research directions include the implementation of encodings for pseudo-Boolean constraints in order to diversify the search for weighted problems.

PWBO is a non-deterministic parallel partial MaxSAT solver. Even though PWBO is able to improve the performance of sequential MaxSAT solvers, it cannot be used in application domains that require reproducible results. The next chapter presents a deterministic version of PWBO that can be used in such domains.

5

Deterministic Parallel MaxSAT

Despite being able to improve the performance of sequential MaxSAT solvers, current parallel MaxSAT solvers cannot be used in application domains that require reproducible results. For example, if we use a parallel MaxSAT solver in software verification [JM11, CSMSV10], different runs can report different bugs when verifying the same program. This behavior is unacceptable for the end user and restrains the use of parallel MaxSAT solvers for software verification applications.

Currently, a deterministic behavior of PWBO is not guaranteed, namely in terms of the solution it provides for the MaxSAT formula. This is a severe limitation on the use of PWBO. Therefore, this chapter explores deterministic approaches for the portfolio version of PWBO and is organized as follows. First, we present a study of the non-deterministic behavior of PWBO. Next, we present deterministic approaches for PWBO. Different synchronization methods are proposed, namely, standard, period and dynamic synchronization. Finally, we analyze the deterministic solver with respect to its variation and how it compares against the non-deterministic version of PWBO.

5.1 Non-Deterministic Behavior of PWBO

PWBO exhibits a non-deterministic behavior mainly due to the cooperation between threads. In this section a study of the non-deterministic behavior is presented. All experiments presented in the remainder of this chapter were performed on two AMD Opteron 6172 processors (2.1 GHz with 64 GB of RAM) running Fedora Core 13 with a timeout of 1,800 seconds (wall clock time). Unless stated otherwise, the set of benchmarks corresponds to the partial MaxSAT instances from

Table 5.1: Example of the non-deterministic behavior of PWBO

Instance	#Models	Avg. Time	Std. Dev.	Δ
		Industrial		
f20c10b_001_area_delay	10	293.82	84.38	28.72
simp-ibd_50.04	10	186.94	22.07	11.81
SU1__simp-genos.haps.29	10	76.16	16.04	21.06
ii32b1	10	72.72	13.57	18.66
3ebx_.6ebx_.g.wcnf.t	2	345.14	78.86	22.85
		Crafted		
frb30-15-1.partial	1	452.52	42.71	9.44
max_clq_150-15-7152-1.clq	1	688.68	100.38	14.58
keller4.clq	1	162.45	8.34	5.13
cnf3.150.500.646984.cnf	7	210.12	41.13	19.58
kbtree9_7_3_5_30_5.wcsp	3	137.37	59.20	43.1

the industrial and crafted categories of the MaxSAT Evaluation of 2011[1]. Note that PWBO is best suited for industrial benchmark instances, as it performs better for industrial than for crafted benchmark instances. Nevertheless, the crafted benchmarks were also included in our evaluation in order to increase our pool of benchmarks. The results for the non-deterministic parallel solver were obtained by running the solver ten times. Similarly to what is done when analyzing randomized solvers, the median time was taken into account. The deterministic versions were run once since the variation of their run times is small. All solvers were run with 4 threads.

To study the non-deterministic behavior of PWBO, we used the portfolio version of PWBO (PWBO-P-T4) presented in the previous chapter. For simplicity, in the remainder of this chapter we will denote this version as PWBO.

Table 5.1 illustrates the non-deterministic behavior of PWBO (version 2.0, 2012[2]). Note that the instances mentioned in the table were solved in all runs of the non-deterministic solver. The first column shows the name of the instance and the second column presents the number of different models that were found in ten runs of the solver. The following columns show the average time in seconds (Avg. Time), standard deviation (Std. Dev.) and the coefficient of variation (Δ) of the run times.

Definition 5.1 (Coefficient of Variation)**.** *The coefficient of variation is a normalized measure of dispersion and is given by* $\Delta = \frac{\mu}{\sigma} \times 100$*, where* μ *is the average time and* σ *the standard deviation.*

These examples show that some instances may have a high coefficient of variation, i.e. their

[1] `http://maxsat.ia.udl.cat/`

[2] Version 2.0 of PWBO has small algorithmic improvements with respect to the first version of PWBO. For example, the underline SAT solver (MiniSAT 2.0 [ES03]) is improved by changing the restart strategy and the polarity heuristic to Luby restarts [LSZ93] and phase saving [PD07], respectively.

Table 5.2: Overview of the non-deterministic behavior of PWBO

	#I	#Solved	Avg. #M	Avg. Δ
Industrial	504	405	7.52	20.77
Crafted	372	240	2.28	22.11
Total	876	645	5.57	21.12

run times may vary significantly between runs. The number of different models is also high for most industrial instances, for which a different model is usually found in each run.

For a better understanding of the average number of different models and the average coefficient of variation, we analyzed the entire set of industrial and crafted benchmarks. Table 5.2 shows an overview of the non-deterministic behavior of PWBO. This table presents the number of instances for each benchmark set (#I), the number of solved instances by PWBO (#Solved), the average number of different models (Avg. #M) and the average coefficient of variation (Avg. Δ) of the run times. The coefficient of variation can have high values for small run times. To reduce the noise in this dispersion measure, when computing the average coefficient of variation we did not consider instances that were solved by all runs of the solver in less than 10 seconds.

The average number of different models is particularly high for industrial benchmarks, since for each instance we find on average more than 7 different models in 10 runs of the solver. On the other hand, for many of the crafted benchmarks instances the same solution was found on all runs.

The average coefficient of variation is around 21% and it is similar for both industrial and crafted benchmarks. Even though this variation is high, it is similar to the variation of portfolio parallel SAT solvers on solving satisfiable instances [HJS09b]. As expected, these results support the idea that PWBO exhibits a high non-determinism on both running times and models found. Therefore, if PWBO is to be used in practice, it is necessary to build a deterministic version of the solver, as end users must be able to replicate the application behavior for the same input.

5.2 Deterministic Solver

Recently, Hamadi et al. [HJPS11] proposed the first deterministic parallel SAT solver. The deterministic solver only exchanges information between threads at fixed points during the search. These points are denoted as synchronization points. Whenever a thread reaches a synchronization point, it waits until the remaining threads reach the same point. Afterwards, when all threads

reach the synchronization point, they exchange learned clauses. This synchronization guarantees the determinism of the cooperation between threads. The use of synchronization points for deterministic parallel SAT solving motivated the approach presented in this section for building a deterministic parallel MaxSAT solver.

This section presents the first deterministic parallel MaxSAT solver that ensures the reproducibility of results. Our deterministic solver is built on top of the portfolio version of PWBO (version 2.0, 2012) described in the previous chapter. The goal of the deterministic solver is to be able to reproduce the same results on solving each problem instance by ensuring the following constraints: (i) the solution reported by the solver is always the same and (ii) the search performed by each thread is also the same.

Figure 5.1 exemplifies an execution of the deterministic solver with 4 threads (but it can be easily generalized to any number of threads). In this example, threads t_1 and t_2 search on the lower bound value of the optimal solution, while threads t_3 and t_4 search on the upper bound value of the optimal solution. Each thread begins by performing its search as in the non-deterministic version described in the previous chapter. Every time a clause is learned, it is exported to the remaining threads. However, in the deterministic solver, learned clauses are only incorporated in other threads at synchronization points. This contrasts to the non-deterministic version where learned clauses can be imported on-the-fly.

When a thread that is searching on the lower bound finds an unsatisfiable core, it stops the search and proceeds to the synchronization point. As can be seen in Figure 5.1, before reaching the synchronization point each thread exports the unsatisfiable core that was found during the last period. A period corresponds to the search done between two consecutive synchronization points. Note that if an unsatisfiable core has not been found in the last period, then only learned clauses are exported.

Remember that to each unsatisfiable core there is an associated cost that corresponds to an increase in the lower bound value and is used by the unsatisfiability-based algorithm to iteratively relax the MaxSAT formula [MMSP09]. Consider k threads performing lower bound search. At a synchronization point, all unsatisfiable cores that were found in the last period are analyzed. Our goal is to import the unsatisfiable core that corresponds to the largest increase in the lower bound value. If two threads find an unsatisfiable core that corresponds to the same increase in the lower bound value, then the unsatisfiable core with the smallest size is imported by all threads. If there are two unsatisfiable cores that have the same size, then ties are broken considering the threads

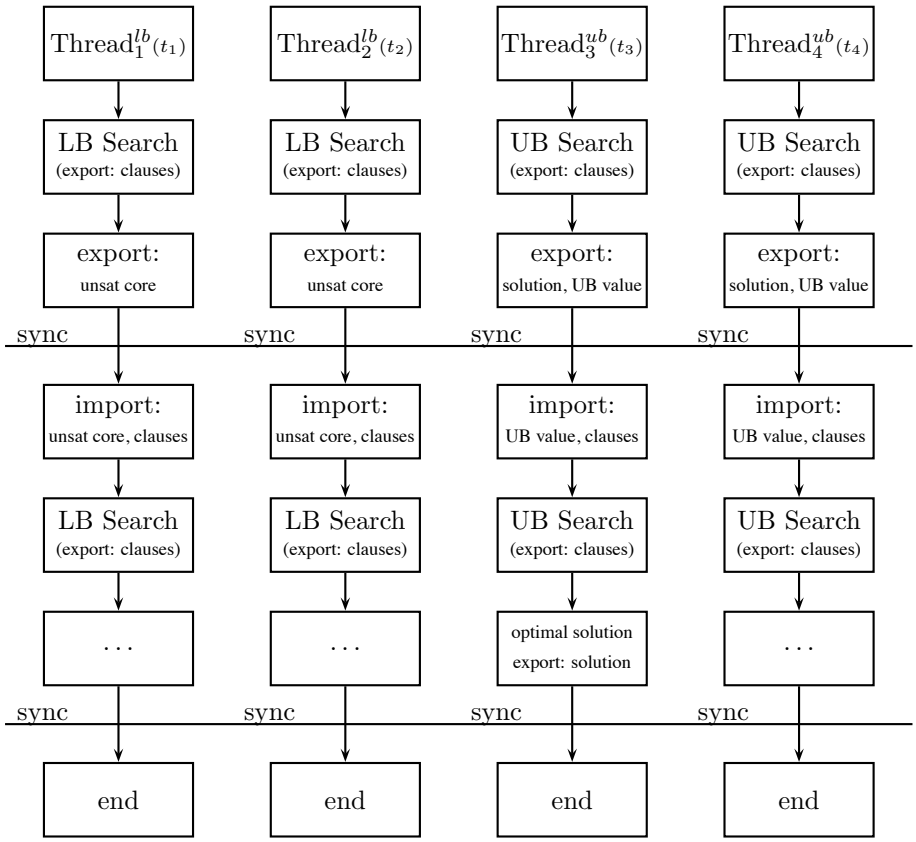


Figure 5.1: Execution of the deterministic solver based on synchronization points

identifiers in increasing order. For example, consider in Figure 5.1 that thread t_1 and thread t_2 find an unsatisfiable core with the same cost and size. After the synchronization point, thread t_1 does not import the unsatisfiable core from thread t_2. On the other hand, thread t_2 discards the unsatisfiable core that was found in the last period and imports the unsatisfiable core exported by thread t_1. Notice that, similarly to the non-deterministic version, all threads that are searching on the lower bound always have the same unsatisfiable cores after a synchronization point.

Threads that are searching on the upper bound export their best solution and the corresponding upper bound value before reaching the synchronization point. At a synchronization point, each thread imports the smallest upper bound value between all threads. As a result, all threads that are searching on the upper bound will have the same upper bound value after the synchronization point.

Learned clauses are also imported at synchronization points. Each thread imports the learned clauses that were exported by the remaining threads since the last synchronization point. Note that

threads searching on the lower bound can also selectively import learned clauses from threads that are performing an upper bound search. The converse is also true [MML11a]. In order to guarantee a deterministic behavior, learned clauses must be imported in the same order. Therefore, in this case, learned clauses are imported using an ascending order with respect to the threads identifiers.

In addition, we must also guarantee the determinism of the solution reported. For a given problem instance, the variable assignments of the optimal solution that the solver outputs must always be the same for all runs of the solver. Every time a new solution is exported, it is only recorded if its corresponding value is smaller than the best value found so far. If the new solution has the same value as the current best value, then the thread identifier is used to decide if the new solution is recorded or not. If the identifier of the exporting thread is smaller than the identifier of the thread where the previous solution was found, then the new solution is recorded. Otherwise, it is discarded. Finally, a thread stops when it proves optimality. However, the remaining threads are only terminated when their next synchronization point is reached. This is done to guarantee the determinism of the reported solution, since new optimal solutions may be found in the remaining running threads. For example, in Figure 5.1 thread t_3 finds an optimal solution but it does not immediately terminate the solver. Instead, it waits until the remaining threads reach the next synchronization point to guarantee that no other optimal solutions have been found in the meantime.

Definition 5.2 (Standard Synchronization [MML12b])**.** *The standard synchronization denotes the kind of synchronization described in this section. In this kind of synchronization, the threads that are searching on the lower bound reach a synchronization point every time a new unsatisfiable core is found.*

5.3 Standard Synchronization

As described in the previous section, the deterministic solver is based on the existence of synchronization points. However, a deterministic measure must be used to define synchronization points. Otherwise, the solver would remain non-deterministic. In the context of parallel SAT solving, Hamadi et al. [HJPS11] propose to use the number of conflicts as a measure for defining the synchronization points. A similar approach can be used for parallel MaxSAT. A simple strategy is to use a static number of conflicts to determine when a thread should reach a synchronization point. For example, each thread has to perform k conflicts before reaching the next synchronization

Table 5.3: Standard synchronization using different intervals

Industrial			
#Conflicts	#Solved	Avg. #Sync	Time
100	400	398.06	26,618.92
1,000	400	43.67	27,673.70
10,000	397	5.47	32,647.13
Crafted			
#Conflicts	#Solved	Avg. #Sync	Time
100	234	770.91	16,757.88
1,000	236	86.39	16,151.80
10,000	236	9.04	14,643.81
Total			
#Conflicts	#Solved	Avg. #Sync	Time
100	634	536.94	43,376.80
1,000	636	59.59	43,825.50
10,000	633	6.80	47,290.94

point.

Table 5.3 shows the impact of choosing different values of k conflicts, namely 100, 1,000 and 10,000 conflicts. For each k, Table 5.3 presents the number of instances solved by the deterministic solver (#Solved), the average number of synchronization points per instance (Avg. #Sync) and the total CPU time in seconds (Time) used by instances that were solved by all deterministic versions.

For $k = 100$, the average number of synchronization points is nearly 537. For larger values of k, the number of synchronization points naturally decreases. For example, for $k = 1,000$ the solver has on average nearly 60 synchronization points, and this value further decreases to only 7 synchronization points when k is increased to 10,000 conflicts. For the values of k tested, $k = 1,000$ conflicts led to the best performance since it is able to solve more instances. There is a trade off between having a large number of synchronization points and the frequency that the information is exchanged between threads. If k is too large, then the number of synchronization points is low but learned clauses and information regarding the bounds is exchanged less frequently. On the other hand, if k is too small then the number of synchronization points is too high which may deteriorate the performance of the solver. Since the value $k = 1,000$ conflicts led to the best performance, this value will be used in all deterministic versions of the solver presented in this section.

Other deterministic measures could be used instead of the number of conflicts. We have tested a few other measures (e.g. number of propagations, number of decisions) but the results were

Table 5.4: Percentage of idle time using standard synchronization

	#Conflicts		
	100	1,000	10,000
	% Idle	% Idle	% Idle
Industrial	43.61	47.90	53.11
Crafted	43.73	44.42	53.35
Total	43.66	46.62	53.18

similar to the ones presented in Table 5.3.

Table 5.4 shows the percentage of idle time for the different values of k Conflicts.

Definition 5.3 (Percentage of Idle Time)**.** *The percentage of idle time is given by the ratio between the sum of CPU time that was not used by the solver on all instances and the sum of available CPU time for those instances.*

Example 5.1. *Assume a solver took 100 seconds (wall clock time) to solve a given instance. Furthermore, assume that 4 threads worked with a ratio of 2.5 (out of 4), i.e. on average 2.5 threads were always working, whereas 1.5 threads were always idle. For this instance, the solver used* $2.5 \times 100 = 250$ *CPU seconds. However, if the 4 threads were always working then it would be possible to use 400 CPU seconds. Therefore, the available CPU time is 400 seconds and the CPU time that was not used by the solver is 150 seconds. Hence, the percentage of idle time for this instance is* $\frac{150}{400} \times 100 = 37.5\%$.

Results in Table 5.4 show that the percentage of idle time increases with the value of k. As previously mentioned, for larger values of k there are less synchronization points. However, the time that each thread needs to wait on those synchronization points increases substantially. Therefore, there is a trade off between having a large number of synchronization points and the time that each thread needs to wait at each synchronization point.

5.4 Period Synchronization

For problem instances with a large number of unsatisfiable cores, the standard synchronization may result in high idle times since at most one unsatisfiable core can be found within each period.

Example 5.2. *Consider a given instance having an optimal solution with value 100. Assume that the lower bound search is able to solve this instance very quickly by finding 100 unsatisfiable cores, each with weight 1, whereas the upper bound search is unable to solve this instance. In the*

standard synchronization approach, if we use $k = 1,000$ conflicts then every time an unsatisfiable core is found one has to wait for the threads searching on the upper bound to reach 1,000 conflicts. In this case, the lower bound threads would have to wait for the upper bound threads to reach more than 100,000 conflicts before the formula is solved. This can require a prohibitive amount of time which is a critical issue in the standard synchronization approach.

An alternative approach is to synchronize only the threads that are searching on the lower bound when they also reach the number of conflicts k that defines the length of the period. However, notice that in this case more than one unsatisfiable core may be found between two synchronization points.

In this new approach, once all threads reach a synchronization point, we analyze which thread has the largest lower bound value. If two threads have the same lower bound value, then the thread with the smallest identifier is chosen. In order to synchronize the lower bound threads, the chosen thread will export the unsatisfiable cores that were found in the last period to the remaining threads that are searching on the lower bound.

Example 5.3. *Consider that after synchronization point p, threads t_1 and t_2 have working formulas φ_{t_1} and φ_{t_2}, respectively. Moreover, consider that when the synchronization point $p+1$ is reached these threads have the same lower bound value. Thread t_1 does not discard the unsatisfiable cores that were found between synchronization points p and $p+1$. On the other hand, thread t_2 discards the unsatisfiable cores that were found between synchronization points p and $p+1$ and imports the unsatisfiable cores from thread t_1 to the formula φ_{t_2}. Similarly to the standard synchronization approach, after the synchronization point $p+1$ both threads have the same lower bound value.*

Definition 5.4 (Period Synchronization [MML13b])**.** *The period synchronization denotes the kind of synchronization where the threads that are searching on the lower bound only stop at the end of each period.*

The main difference between standard and period synchronization is that in the standard synchronization at most one unsatisfiable core is found during each period, whereas in the period synchronization it is possible to find a larger number of unsatisfiable cores in a single period.

Table 5.5 shows the results for the deterministic solver using period synchronization with $k = 1,000$ conflicts. The new synchronization approach and the standard synchronization solve the same number of instances. However, the average number of synchronization points decreases

Table 5.5: Deterministic solver using period synchronization

	#Sol.	Avg. #Sync	Time	% Idle
Industrial	400	40.59	24,379.22	43.70
Crafted	236	82.09	15,994.93	40.97
Total	636	56.05	40,374.15	42.62

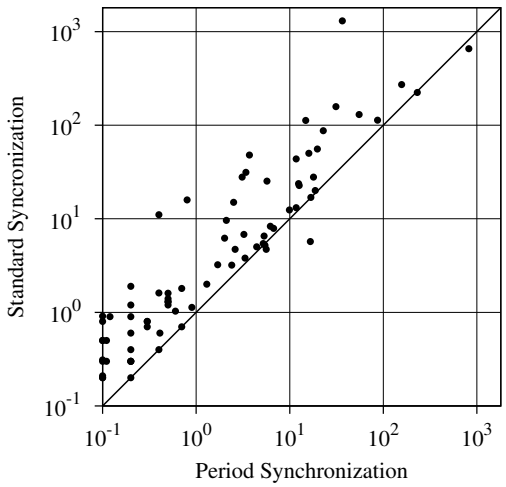


Figure 5.2: Run times of standard synchronization and period synchronization on instances solved by the lower bound approach

from 60 periods in standard synchronization to 56 in period synchronization.

Similarly, the percentage of idle time also decreases from 47% to 42%. Finally, the period synchronization has a better performance since it is able to solve the same set of instances requiring less time than the standard synchronization approach.

The main difference between the different synchronization techniques lies on the instances that are solved using lower bound search. Figure 5.2 compares the standard synchronization with the period synchronization on instances that were solved optimally by the lower bound search on both deterministic solvers, which corresponds to a subset of 87 instances. For most of these instances, the period synchronization clearly outperforms the standard synchronization. Note that if the optimal value is small then the standard synchronization may outperform the period synchronization since the idle time due to the synchronization at each core will not be significant. However, Figure 5.2 shows that this case is not common and is restricted to only a few outliers.

5.5 Dynamic Synchronization

The main problem of a static strategy using a fixed number of conflicts k to define the length of a period is that different threads have different search behaviors and reach k conflicts at very different times. Therefore, using a static strategy may lead to high idle times at each synchronization point for the faster threads. This problem is further accentuated in parallel MaxSAT since the size of the formula can differ substantially between threads. For example, threads that search on the upper bound of the optimal solution and use CNF encodings to encode the constraint on the upper bound value may have a formula that is several times larger than working formulas in other threads. An alternative approach that tries to minimize idle times is to use a dynamic synchronization strategy.

Definition 5.5 (Dynamic Synchronization [MML13b])**.** *The dynamic synchronization denotes the kind of synchronization where each thread dynamically updates the number of conflicts that is required in order to reach the next synchronization point.*

In the context of parallel SAT solving, Hamadi et al. [HJPS11] proposed to use the number of clauses learned in each thread for dynamically updating the necessary number of conflicts to reach the next synchronization point. A similar approach could also be used for parallel MaxSAT. Note that in deterministic parallel SAT all threads are initially run on the same formula. Therefore, all threads are initially working on a formula with the same size. However, as previously mentioned, this is not the case for our deterministic parallel MaxSAT solver. As a result, we propose to take into account the number of clauses and the number of learned clauses in each thread.

Let φ_i and ϕ_i denote the set of initial clauses and the set of learned clauses in the working formula of thread i, respectively. Consider that at synchronization point p, thread i has $|\varphi_i + \phi_i|$ clauses. Let m be the maximum number of clauses between all threads and k the number of conflicts to reach the first synchronization point. The number of conflicts that thread i needs to reach the next synchronization point is given by:

$$sync_i = \quad \lceil k + (1 - \tfrac{|\varphi_i + \phi_i|}{m}) \times k \rceil \tag{5.1}$$

Threads that have more clauses will have smaller periods, whereas threads that have less clauses will have larger periods. The goal is to balance the number of conflicts required by each thread to reach the next synchronization point in an attempt to reduce the idle time of each thread. This dynamic synchronization is based on the size of the formula of each thread and is denoted in the remainder of the chapter by *dyn-size*. Equation (5.1) bounds the number of conflicts for the next

Table 5.6: Deterministic solver using dynamic synchronization

	Industrial			
	#Solved	Avg. #Sync	Time	% Idle
dyn-size	401	36.81	23,758.07	38.58
dyn-k-size	400	32.78	26,508.76	33.51
	Crafted			
	#Solved	Avg. #Sync	Time	% Idle
dyn-size	235	77.94	16,049.94	35.01
dyn-k-size	236	60.49	17,897.29	33.46
	Total			
	#Solved	Avg. #Sync	Time	% Idle
dyn-size	636	52.13	39,808.01	37.09
dyn-k-size	636	43.10	44,406.05	33.49

synchronization point between k and $2k$. However, the size of the formula in some threads may be several times larger than the size in other threads. Therefore, a possible issue is that increasing the period to $2k$ may not be enough to balance the number of conflicts between the different threads. Note that this hardly occurs in parallel SAT since the size of the formulas is very similar.

An alternative approach to balance the periods between threads it to consider the ratio between the sizes of the formulas. This can be achieved using the following equation:

$$sync_i = \lceil \frac{m}{|\varphi_i + \phi_i|} \times k \rceil \tag{5.2}$$

Note that equation (5.2) does not have an upper bound on the number of conflicts to reach the next synchronization point. The thread with the largest formula will have a period of length k, whereas the thread with the smallest formula can have a period several times larger than k. This kind of dynamic synchronization is denoted by *dyn-k-size*. The goal of this dynamic synchronization is to further reduce the idle time by taking into consideration that the size of the formulas of each thread can be significantly different.

Table 5.6 compares the *dyn-size* and the *dyn-k-size* versions of the deterministic solver. Both versions solve the same number of instances. However, *dyn-size* is more efficient than *dyn-k-size*. For instances that are solved by all deterministic versions, *dyn-size* is much faster than *dyn-k-size*. The idle percentage of both versions is lower than the idle percentage of the static versions. For example, *dyn-k-size* has an average idle time of 33%, whereas the deterministic solver that uses period synchronization has an average idle time of 43%. Overall, *dyn-k-size* has the lowest percentage of idle time among the tested deterministic versions. However, we can observe that reducing the idle time does not ensure the performance of the solver to improve. This is due to the

Table 5.7: Percentage of idle time per thread

	dyn-size				dyn-k-size			
	t_1	t_2	t_3	t_4	t_1	t_2	t_3	t_4
Industrial	42.61	40.16	11.14	50.83	34.55	31.98	19.95	39.84
Crafted	42.73	36.87	29.09	27.89	39.71	32.30	37.15	19.53
Total	42.66	38.74	18.88	40.93	36.63	32.11	26.88	31.65

heterogeneous performance of each thread. It was noticed that the thread that performs upper bound search using a CNF encoding is more efficient than the remaining threads. Note that the size of the formula of this thread is usually larger than the remaining threads.

Table 5.7 presents the percentage of idle time per thread. Threads t_1 and t_2 search on the lower bound of the optimal solution, while threads t_3 and t_4 search on the upper bound of the optimal solution. Moreover, threads t_1, t_2 and t_3 encode the new constraints added at each iteration using CNF encodings, whereas t_4 is able to deal natively with cardinality constraints.

Increasing the idle time of thread t_3 has a significant impact on the performance of the solver. This explains why *dyn-size* outperforms *dyn-k-size* even though it has an overall higher percentage of idle time. Due to its performance, in the remainder of the dissertation the deterministic version *dyn-size* will be used as the dynamic deterministic solver.

5.6 Analysis of the Deterministic Solver

As seen in section 5.1, the non-deterministic solver exhibits a high variation in both the run time and the number of models found. Even though the deterministic solver always performs the same search at each run, it may exhibit a small variation of run time due to the synchronization procedures.

Table 5.8 shows the average number of models (Avg. #M) and the coefficient of variation (Avg. Δ) of the run times for ten runs of the dynamic deterministic solver. The deterministic solver always finds the same model within the same number of synchronization points. However, the average coefficient of variation is less than 7%. Considering that the same model is always found and that the variation is small, this shows that our deterministic solver is a stable tool to be used by practitioners in their applications.

Table 5.9 compares the non-deterministic version with the deterministic versions that use standard synchronization, period synchronization and dynamic synchronization. All deterministic versions were run with an initial value of $k = 1,000$ conflicts.

Table 5.8: Variation of the deterministic solver

	Avg. #M	Avg. Δ
Industrial	1.00	7.74
Crafted	1.00	4.20
Total	1.00	6.88

Table 5.9: Comparison between the non-deterministic and deterministic solvers

	Industrial		Crafted		Total	
	#Solved	Speedup	#Solved	Speedup	#Solved	Speedup
Non-Deterministic	405	1.00	241	1.00	646	1.00
Standard	400	0.77	236	0.74	636	0.76
Period	400	0.88	236	0.75	636	0.83
Dynamic	401	0.90	235	0.75	636	0.84

All deterministic versions solve the same number of instances. On the other hand, the non-deterministic version is able to solve more 10 instances than the deterministic versions. The differences in the performance of the deterministic versions are clear for the industrial benchmarks. However, for the crafted benchmarks the performance is similar for all deterministic versions. Note that PWBO is more efficient for industrial benchmarks. Hence, if the deterministic version is going to be used by practitioners it will most likely be on industrial benchmarks. Even though the non-deterministic solver is more efficient, the run times of the non-deterministic solver are comparable to the ones of the deterministic solvers.

5.7 Summary

Parallel MaxSAT solvers can improve the performance of sequential MaxSAT solvers. However, they cannot be used in application domains that require reproducible results. This provides a strong stimulus to build a deterministic parallel MaxSAT solver that is able to reproduce the same results on solving each problem instance. In this context, this chapter presents the first deterministic parallel MaxSAT solver. Different approaches for thread synchronization are studied, namely, standard, period and dynamic synchronization. Dynamically changing the number of conflicts that each thread requires to reach a synchronization point improves the performance of the deterministic solver. Moreover, the dynamic synchronization also reduces the idle time of the solver. An analysis of the performance of our deterministic parallel MaxSAT solver with the non-deterministic versions shows that the non-deterministic solver can only solve a few more

instances. Moreover, for instances that are solved by both solvers, the deterministic solver exhibits a performance that is similar to the non-deterministic version.

Evaluating heuristics can be a hard task in parallel MaxSAT due to the non-deterministic behavior of parallel MaxSAT solvers. Therefore, another application of deterministic solvers is to the evaluation of the performance of heuristics since these solvers provide a parallel experimental setup where the only variation is the heuristic. In the next chapter, the deterministic solver presented in this chapter is used for a fair evaluation between the different clause sharing heuristics.

6

Clause Sharing Heuristics

In parallel MaxSAT solving, sharing learned clauses is expected to improve the performance of a parallel solver. However, not all learned clauses should be shared since it could lead to an exponential blow up in memory and to sharing many irrelevant clauses. A clause is considered irrelevant if it never becomes unsatisfied or unit, which means that it does not help in pruning the search space. The problem of determining if a shared clause will be useful in the future remains challenging and in practice heuristics are used to select which learned clauses should be shared.

Definition 6.1 (Shared Learned Clauses)**.** *Shared learned clauses correspond to learned clauses that were exported by a thread and were given to other threads. The importing thread can then decide if it incorporates the shared learned clause into its context or not.*

Definition 6.2 (Exporting and Importing Threads)**.** *An exporting thread is a thread that sends learned clauses to other threads. On the other hand, an importing thread is a thread that receives learned clauses sent by an exporting thread.*

Clause sharing heuristics can be divided into the following three categories: (i) static, (ii) dynamic and (iii) freezing. The static heuristics share learned clauses within a given cutoff, whereas the dynamic heuristics adjust this cutoff during the search. Alternatively, the freezing heuristics temporarily delay the incorporation of shared clauses until they are expected to be useful in the context of the importing thread.

This chapter is organized as follows. First, we describe the different heuristics for clause sharing. Next, we use the deterministic solver described in the previous chapter to compare the impact of each clause sharing heuristic in the performance of the solver.

6.1 Static Heuristics

The static heuristics are the most used heuristics for clause sharing since they are simple but still efficient in practice. The following measures are used in these heuristics:

- *Size*: the clause size is given by the number of literals in it. Small clauses are expected to be more useful than larger clauses. Clause size was originally used as a measure to select which learned clauses should be kept by the SAT solver [MSS96, MSS99]. More recently, it has been adapted by parallel SAT solvers (e.g. [HJS09b]) and parallel MaxSAT solvers [MML12e].

- *Literal Block Distance* (LBD) [AS09]: the literal block distance corresponds to the number of different decision levels involved in a clause. Clauses with small LBD are considered as more relevant.

6.2 Dynamic Heuristics

It has been observed that the size of learned clauses tends to increase over time. Consequently, in parallel solving, any static limit may lead to halting the clause sharing process. Therefore, to continue sharing learned clauses it is necessary to dynamically increase the limit during search. In the context of parallel SAT solving, Hamadi et al. [HJS09a] proposed the following dynamic heuristic. At every k conflicts (corresponding to a period α) the number of shared learned clauses (s) is evaluated between each pair of threads ($t_i \rightarrow t_j$) according to the following heuristic:

$$\lim_{t_i \rightarrow t_j}^{\alpha+1} = \begin{cases} \text{if s} < \text{m (sharing is small):} & \lim_{t_i \rightarrow t_j}^{\alpha} + \text{quality}_{t_i \rightarrow t_j}^{\alpha} \times \frac{b}{\lim_{t_i \rightarrow t_j}^{\alpha}} \\ \text{if s} \geq \text{m (sharing is large):} & \lim_{t_i \rightarrow t_j}^{\alpha} - (1 - \text{quality}_{t_i \rightarrow t_j}^{\alpha}) \times a \times \lim_{t_i \rightarrow t_j}^{\alpha} \end{cases}$$

where a and b are positive constants and the value of $\text{quality}_{t_i \rightarrow t_j}^{\alpha}$ corresponds to the quality of shared learned clauses that were exported from t_i and imported by t_j. The number of shared learned clauses in a period k is given by s. If s is less than a given m, then the sharing in period k is considered to be small. Otherwise, if s is greater than or equal to m, then the sharing in period k is considered to be large.

Definition 6.3 (Clause Quality)**.** *A shared learned clause with n literals is said to have quality if at least half of its literals are active when the learned clause is exported.*

Definition 6.4 (Active Literal [HJS09a]). *A literal is active if its VSIDS heuristic [ZMMM01] score is high, i.e. if it is likely to be chosen as a decision variable in the near future. Consider that m corresponds to the VSIDS score of the variable with the largest VSIDS score. A literal l is active, if its VSIDS score is larger than m/2.*

Definition 6.5 (Quality of Sharing [HJS09a]). *] The quality of sharing between each pair of threads ($t_i \rightarrow t_j$) is given by the following heuristic:*

$$quality^{\alpha}_{t_i \rightarrow t_j} = \frac{q}{s}$$

where q is the number of quality shared learned clauses and s is the total number of shared learned clauses in the period α.

If the quality of sharing is high then the increase (decrease) in the size limit of shared learned clauses will be larger (smaller). The idea behind this heuristic is that the information recently received from a thread t_i is qualitatively linked to the information to be received from the same thread t_i in the near future. In our experimental setting for parallel MaxSAT, we have selected $a = 0.125, b = 8$ and $\alpha = 3000$ conflicts. The throughput m at each period is set to 750, i.e. if a thread t_j receives less than 750 shared learned clauses in the period α then, it increases the limit of the size of shared clauses. Otherwise, this limit is decreased. These parameters are similar to the ones used by Hamadi et al. [HJS09a].

6.3 Freezing Heuristics

There are possible drawbacks to importing clauses shared by other threads. One drawback is that the newly imported clauses may be irrelevant in the context of the importing thread. Another possible drawback is that the exploration of the search space may be influenced in such a way that the search becomes more closely related to the exploration being performed in the thread from which the clauses originated. As a result, the diversification of the exploration of the search space is decreased by shifting the context of the current search in the importing thread.

Our motivation for the freezing heuristic is to only incorporate shared clauses when they are expected to be useful in the near future. For that, the decision to incorporate new learned clauses shared by other threads must take into consideration the current search context where these clauses are to be integrated. As a result, these new clauses should improve the efficiency of the search

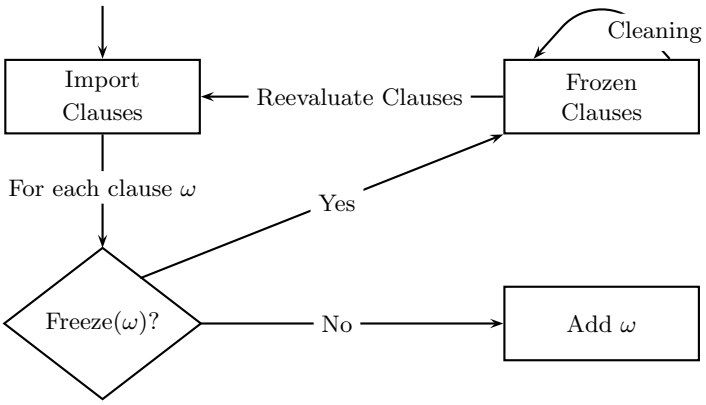


Figure 6.1: Freezing procedure for sharing learned clauses

being carried out as they do not imply a major change to the search context of the receiving thread.

Figure 6.1 illustrates the freezing procedure [MML12c]. Each imported learned clause ω is evaluated to determine if it will be frozen or added to the solver. If ω is frozen then it will be reevaluated later. However, if ω is assigned to the frozen state more than z times then it is permanently deleted. When evaluating ω, our goal is to import clauses that are unsatisfied or that will become unit clauses in the near future. Next, the freezing heuristic is presented. According to the *status* of ω (satisfied, unsatisfied, unit or unresolved), it decides whether ω should be frozen:

- ω is *satisfied*: Let *clevel* denote the current decision level, *level*(ω) the highest decision level of the satisfied literals in ω, *unassignedLits*(ω) the number of unassigned literals in ω and *activeLits*(ω) the number of active literals in ω. If ($clevel - level(\omega) \leq c$) and ($unassignedLits(\omega) - activeLits(\omega) \leq d$) (where c and d are constant values) then ω is imported, otherwise it is frozen. A satisfied clause is expected to be useful in the near future assuming there is no need to backtrack significantly for the clause to become unit. It is also important that the number of unassigned literals is small, otherwise the clause may not become unit in the near future. Active literals are also taken into consideration since they will be assigned in the near future.

- ω is *unsatisfied* or *unit*: ω is always imported;

- ω is *unresolved*: if $unassignedLits(\omega) - activeLits(\omega) \leq d$) then the clause is imported. Otherwise, it is frozen. Similarly to the satisfied case, if the number of unassigned literals is small then ω is likely to be unit in the near future.

Table 6.1: Comparison of the different heuristics for sharing learned clauses

	Industrial		Crafted		Total	
	#Solved	Speedup	#Solved	Speedup	#Solved	Speedup
No sharing	400	1.00	235	1.00	635	1.00
Random	400	1.12	232	0.86	632	1.00
LBD 5	401	1.25	235	1.07	636	1.17
Size 8	401	1.28	235	1.01	636	1.15
Size 32	401	1.24	235	1.01	636	1.13
Dynamic	401	1.38	235	1.02	636	1.20
Freezing	402	1.37	236	1.10	638	1.24

In our experimental setting, we have selected $c = 31, d = 5$ and $z = 7$. In addition, the frozen clauses are reevaluated every 1,000 conflicts (corresponding to a synchronization period of the deterministic solver). These parameters were experimentally tuned. Note that freezing learned clauses has been proposed in the context of deletion strategies for learned clauses in SAT solving [ALMS11]. However, to the best of our knowledge, our solver was the first one to freeze shared clauses in a parallel solving context [MML12c]. Following the recent success of freezing shared clauses in parallel MaxSAT, this idea has also been recently applied to parallel SAT solving [AHJ+12].

6.4 Evaluation of Clause Sharing

To perform a fair comparison between the different heuristics we used the deterministic solver presented in the previous chapter. This solver uses 4 threads and it is based on dynamic synchronization that has synchronization points at every 1,000 conflicts. By using a deterministic solver, one can independently evaluate the gains coming from the use of different heuristics rather than the non-determinism of the solver. All experiments were run on the partial MaxSAT instances from the industrial and crafted categories of the MaxSAT Evaluation of 2011[1], having 497 and 372 instances, respectively. The evaluation was performed on a computer with two AMD Opteron 6172 processors (2.1 GHz with 64 GB of RAM) running Fedora Core 13 with a timeout of 1,800 seconds (wall clock time).

Table 6.1 compares the different heuristics regarding the number of solved instances and the speedup relatively to not sharing any learned clauses. Instances that are easily solved have similar solving times with and without sharing learned clauses. Therefore, the speedup only considers

[1] `http://maxsat.ia.udl.cat/`

instances that take more than 60 seconds to be solved by at least one solver. Note that from the 638 instances that are solved by at least one of the deterministic solvers, 513 instances can be solved by all solvers under 60 seconds. This shows that the majority of the instances is solved quickly and so the impact of sharing learned clauses is less significant. Hence, the speedup presented in Table 6.1 only considers the 125 instances (83 industrial and 42 crafted) that took more than 60 seconds to be solved by at least one solver. By restricting ourselves to this subset of instances, we can have a better understanding of the impact of sharing learned clauses.

To assess the quality of the different heuristics, we have also implemented a random heuristic that randomly decides with probability of 30% to share each learned clause. We denote this heuristic by *Random*. This probability ensures that the number of shared clauses does not grow substantially. Therefore, when using the *Random* heuristic with a probability of 30%, the number of shared clauses is similar to the ones of the remaining heuristics.

The line *No Sharing* in Table 6.1 shows the data for the deterministic solver without sharing learned clauses. Next, the *Random* heuristic is evaluated. Notice that randomly sharing clauses can deteriorate the performance of the solver, making it worse than no sharing clauses. For example, for the crafted category 3 instances less were solved. Moreover, for the crafted category, the random heuristic also deteriorated the performance of the solver. However, for the industrial category, randomly sharing learned clauses actually improved the performance of the solver when compared to not sharing any learned clauses. This shows that for some instances even if sharing is performed randomly, it can still improve the performance of the solver.

Table 6.1 also shows the results for static heuristics, namely using literal block distance with a maximum value of 5 (*LBD 5*), as well as sharing clauses with a size limit of 8 and 32 literals (*Size 8* and *Size 32*). Using a LBD of 5 has similar performances to a size limit of 8 or 32 for the industrial benchmarks. However, for the crafted benchmarks, using a LBD of 5 can be better than using heuristics based on size limits. Note that a clause may be large in size and still have a small LBD. Therefore, this heuristic may find useful learned clauses that are never exported when using a size limit heuristic. Different limits on the size of importing clauses can significantly change the performance of the solver. It was observed that if the limit is too small then the speedup is reduced since not many clauses are shared. On the other hand, if the limit is too large then the speedup is also reduced since many irrelevant clauses are shared. A size limit of 32 is comparable to a size limit of 8, since there are instances where learning larger clauses can be useful. In some cases, a static limit of 8 for clause sharing is too restrictive and does not allow sharing clauses

Table 6.2: Average number of clauses and average size of learned clauses

	Industrial		Crafted	
	Avg. #Clauses	Avg. Size	Avg. #Clauses	Avg. Size
Random	40,686.10	99.57	74,839.00	147.61
LBD 5	20,822.01	12.66	18,317.46	100.65
Size 8	16,903.33	5.41	6,481.11	6.43
Size 32	48,687.91	13.42	39,635.40	18.81
Dynamic	28,496.23	8.57	26,637.68	14.80
Freezing	31,827.38	10.93	34,045.95	16.19

that are important for solving an instance. Nevertheless, any of these heuristics always improve the performance of the solver in terms of run time.

In the dynamic heuristic, each thread starts by importing clauses with at most 8 literals. This cutoff is dynamically adjusted as described in section 6.2. The dynamic heuristic outperforms the static heuristics, being particularly effective on industrial benchmarks.

The freezing heuristic uses a static cutoff of 32. Nevertheless, it differs from the static heuristic of size 32 by delaying the incorporation of the received learned clauses until they are expected to be useful. The freezing heuristic clearly outperforms the static heuristic of size 32. Moreover, the freezing heuristic also outperforms the remaining heuristics since it solves more instances.

Regarding solving times, the dynamic and freezing heuristics show similar performances on solving industrial benchmarks. However, both of these heuristics clearly outperform the remaining ones. For the crafted benchmarks, the freezing heuristic outperforms the remaining heuristics in terms of instances solved and solving times. Even though the impact of sharing learned clauses is higher for industrial benchmarks than for crafted benchmarks, the freezing heuristic still improves the performance of the solver for crafted benchmarks.

Table 6.2 shows the average number of clauses imported by each thread and the average size of the imported clauses. Table 6.3 shows a correlation between the percentage of imported clauses and their size. The results shown in these tables use the 125 instances previously described, i.e. corresponding to the instances that are solved by at least one of the deterministic solvers in more than 60 seconds.

Table 6.3 shows the distribution of learned clauses by size. For each interval $([1,8],]8,16],]16,32]$, $]32,128]$ and $]128,\infty[$), it shows the percentage of shared clauses having a size within the corresponding interval. With no surprise, the random heuristic shares learned clauses with the largest average size since there is no limit on the size of the shared learned clauses. For the industrial

Table 6.3: Distribution of the learned clauses by clause size in percentage

	Industrial				
	$[1,8]$	$]8,16]$	$]16,32]$	$]32,128]$	$]128,\infty[$
Random	13.57	13.86	12.51	31.81	28.24
LBD 5	60.68	23.51	11.09	4.09	0.63
Size 8	100.00	–	–	–	–
Size 32	32.88	34.47	32.65	–	–
Dynamic	58.51	35.24	6.06	0.19	–
Freezing	48.27	31.08	20.65	–	–

	Crafted				
	$[1,8]$	$]8,16]$	$]16,32]$	$]32,128]$	$]128,\infty[$
Random	2.3	6.57	8.64	31.93	50.56
LBD 5	16.35	5.19	0.81	43.75	33.9
Size 8	100.00	–	–	–	–
Size 32	12.46	39.45	48.09	–	–
Dynamic	22.01	51.46	22.1	4.43	–
Freezing	15.93	47.86	36.21	–	–

benchmarks, 31.81% of the shared clauses have size between 32 and 128 and 28.24% of the shared clauses have size larger than 128. For the crafted benchmarks, this situation is further accentuated since 50.56% of the shared clauses have size larger than 128. Moreover, the random heuristic shares, on average, more clauses than the remaining heuristics for the crafted benchmarks. This may explain why the random heuristic deteriorates the performance of the solver for the crafted benchmarks.

The LBD heuristic showed a good performance on crafted benchmarks. For these benchmarks, this heuristic shares clauses with an average size of 100.65. Even though the shared clauses have a large size, these clauses are useful as they improve the solver's performance. Table 6.3 reinforces this idea by showing that 43.75% of the shared clauses have size between 32 and 128 and 33.9% of the shared clauses have size larger than 128.

Using a limit of size 8 or 32 results in similar performances even though the number of shared clauses differs significantly between these two heuristics. For example, when considering the crafted benchmarks with a size limit of 8, each thread imports on average 6,481 learned clauses. On the other hand, if a size limit of 32 is used then each thread imports on average 39,635 learned clauses. The results also show that the average size of learned clauses is larger in the crafted benchmarks than in the industrial benchmarks. Let us analyze the distribution of learned clauses when using a size limit of 32. For the industrial benchmarks, the clauses are evenly distributed between the intervals $[1,8]$, $]8,16]$ and $]16,32]$. However, for the crafted benchmarks we can clearly see that

48.09% of the shared clauses have size between 16 and 32, whereas only 12.46% of the clauses have size between 1 and 8.

In the dynamic heuristic, each thread starts by importing clauses with at most 8 literals. This cutoff is dynamically updated during the search. On average each thread increases the cutoff to 20.14 for industrial benchmarks and 31.73 for crafted benchmarks. Due to this increase, the average number of imported clauses by each thread almost doubled for the industrial benchmarks from the static heuristic with size 8 to the dynamic heuristic that starts with a cutoff of size 8. For the crafted benchmarks, this effect is further accentuated since the number of imported clauses by each thread increases by a factor of 4 when compared to the a static cutoff of size 8. Table 6.3 shows that for industrial benchmarks the dynamic heuristic shared the majority of clauses within the initial cutoff, i.e. it shares learned clauses that have 8 or less literals. However, for the crafted benchmarks, the dynamic heuristic increases the cutoff considerably and 51.46% of the shared clauses have size between 8 and 16.

The freezing heuristic uses a static cutoff of size 32. Nevertheless, not all imported learned clauses are incorporated into the solver. Table 6.2 shows that the average number of imported clauses by each thread is smaller when using the freezing heuristic than when using the static heuristic of size 32. Notice that in the freezing heuristic some imported clauses may be deleted. If a clause is imported and in the following synchronization periods is not considered to be helpful for the solver, then this clause is deleted. For the industrial benchmarks, the freezing heuristic freezes around 60% of the imported clauses. From these instances, more than half are deleted and are never added to the solver. For crafted benchmarks, the freezing heuristic freezes around 54% of the imported clauses and from those around 85% are deleted. Table 6.3 shows that the freezing heuristic imports less clauses with size between 16 and 32 than the static heuristic of size 32. This leads to believe that the freezing heuristic tends to delete clauses with larger size since those clauses are less likely to be useful for the solver.

6.5 Summary

This chapter described different sharing heuristic procedures that were already proposed for parallel SAT solving and integrate them into a deterministic parallel MaxSAT solver. Moreover, a new heuristic based on the notion of freezing is proposed in this chapter. This heuristic delays incorporating the shared learned clauses that were imported from other threads. These clauses are frozen until they are considered relevant in the context of the current search.

An application of deterministic solvers is to evaluate the performance of heuristics since they allow having a parallel experimental setup such that the only variation is the heuristic. Experimental results show that sharing learned clauses in a deterministic portfolio-based MaxSAT solver does not increase significantly the number of solved instances. However, it does allow a considerable reduction of the solving time. Moreover, the new freezing heuristic outperforms all other heuristics both in solving time and number of solved instances. As future work, one should consider aggregating several criteria for clause sharing. Variations on the freezing heuristic can also be devised in order to take into consideration other information from the context of search space being explored in the importing thread.

7

Improving Sequential MaxSAT

Problem partitioning is a well-known technique used for general problem solving and it has already been used in the context of Boolean optimization [Cou96] formulations. The main goal of partitioning is to identify easier to solve subproblems such that partitioning will help to solve the overall problem.

In section 3.2.2, we have presented the unsatisfiability-based MaxSAT algorithms used throughout this dissertation. These algorithms are based on iteratively calling a SAT solver enhanced with the ability of providing a certificate of unsatisfiability. However, one drawback of these algorithms results from the SAT solver returning unnecessary large unsatisfiable cores as certificates. In this chapter, we propose to improve these algorithms by using a new technique based on partitioning the formula. Instead of dealing initially with the whole formula, we start with a smaller formula that is extended at each iteration of the algorithm. The goal is to initially have smaller formulas that enable the SAT solver to provide smaller certificates of unsatisfiability.

This chapter is organized as follows. The next section describes how we can enhance the original unsatisfiability-based algorithm for weighted partial MaxSAT [MMSP09, ABL09], section 3.2.2) with partitioning soft clauses. Next, different partition methods for MaxSAT are presented. Experimental results are presented in section 7.3 and show the effectiveness of the proposed algorithm.

Algorithm 5: Unsatisfiability-based algorithm for weighted partial MaxSAT enhanced with soft partitioning

Input: $\varphi = \varphi_h \cup \varphi_s$
Output: satisfying assignment to φ or UNSAT

```
1  (st, ν, φ_C) ← SAT(φ_h)                          // check if the MaxSAT formula is unsatisfiable
2  if st = UNSAT then
3      return UNSAT
4  γ ← ⟨γ_1, ..., γ_n⟩ ← partitionSoft(φ_s)
5  φ_W ← φ_h
6  while true do
7      φ_W ← φ_W ∪ first(γ)
8      γ ← γ \ first(γ)
9      (st, ν, φ_C) ← SAT(φ_W)
10     while st = UNSAT do
11         min_c ← +∞
12         foreach (ω, w) ∈ (φ_C ∩ φ_s) do
13             if w < min_c then
14                 min_c ← w                          // minimum weight of φ_C
15         V_R ← ∅
16         foreach (ω, w) ∈ (φ_C ∩ φ_s) do
17             V_R ← V_R ∪ {r}                        // r is a new variable
18             (ω_R, w') ← (ω ∪ {r}, min_c)           // relax soft clause
19             if w > min_c then
20                 (ω, w) ← (ω, w − min_c)            // duplicate soft clause
21                 φ_W ← φ_W ∪ {ω_R}
22             else
23                 φ_W ← φ_W \ {ω} ∪ {ω_R}
24         φ_W ← φ_W ∪ {CNF(Σ_{r∈V_R} r ≤ 1)}
25         (st, ν, φ_C) ← SAT(φ_W)
26     if γ = ∅ then
27         return ν                                   // satisfying assignment to φ_W
```

7.1 Partition-based MaxSAT Algorithms

Algorithm 5 illustrates an unsatisfiability-based algorithm for weighted partial MaxSAT enhanced with partitioning soft clauses [MML12d]. The main differences between Algorithm 5 and the original unsatisfiability-based algorithm for weighted partial MaxSAT presented in section 3.2.2 are highlighted. In the original algorithm, φ_W is initialized with $\varphi_h \cup \varphi_s$. This contrasts with the proposed algorithm, where φ_W is initialized with φ_h and one partition of soft clauses is added at each iteration.

Algorithm 5 starts by checking if the MaxSAT instance φ is satisfiable by calling a SAT solver only with hard clauses φ_h. Next, the set of soft clauses φ_s is split into a list of partitions (line 4) such that each soft clause is assigned to one partition. Initially, the working formula only considers

the hard clauses φ_h (line 5). At each iteration, a partition γ_i of soft clauses is added to the working formula (line 7) and removed from the partition list γ (line 8). A SAT solver is then applied to φ_W, returning a triple (st, ν, φ_C) where st denotes the outcome of the solver: SAT or UNSAT. While the outcome is UNSAT, φ_C contains the unsatisfiable core identified by the SAT solver and the unsatisfiable core is relaxed as in the original algorithm (Algorithm 4, section 3.2.2). Next, the SAT solver is applied to the modified working formula (line 25). After a given number of relaxations, the working formula becomes satisfiable[1] and a new partition of soft clauses is added to the working formula. If there are no more partitions in γ, then the solver found an optimal solution to the original MaxSAT formula (line 27). Next, we present an example of the execution of Algorithm 5.

Example 7.1. *Consider a weighted partial MaxSAT formula $\varphi = \varphi_h \cup \varphi_s$ such that:*

$$\begin{aligned} \varphi_h = &\quad \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3]\} \\ \varphi_s = &\quad \{(x_1, 100), (x_3, 100), (x_2 \vee \bar{x}_1, 1), (\bar{x}_3 \vee x_1, 1)\} \end{aligned} \tag{7.1}$$

Moreover, consider that we are given two partitions of soft clauses, γ_1, γ_2, where $\gamma_1 = \{(x_1, 100), (x_3, 100)\}$ and $\gamma_2 = \{(x_2 \vee \bar{x}_1, 1), (\bar{x}_3 \vee x_1, 1)\}$.

All hard clauses together with the soft clauses from γ_1 are given to the SAT solver. The formula is unsatisfiable and an unsatisfiable core φ_C is identified. Suppose $\varphi_C = \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], (x_1, 100), (x_3, 100)\}$. The formula is relaxed as in the original unsatisfiability-based algorithm. As a result, the formula is updated as follows:

$$\begin{aligned} \varphi_W = &\quad \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], \\ &\quad (x_1 \vee r_1, 100), (x_3 \vee r_2, 100), \\ &\quad [\mathit{CNF}(r_1 + r_2 \leq 1)]\} \end{aligned} \tag{7.2}$$

The resulting formula is now satisfiable and the SAT solver returns the following satisfying assignment $\nu = \langle \bar{x}_1, x_2, x_3, r_1, \bar{r}_2 \rangle$. Even though the SAT solver returned a satisfying assignment, there are still partitions of soft clauses that were not yet added to the working formula. Therefore, the next partition (γ_2) is now added to φ_W and the resulting formula is given to the SAT solver. This formula is unsatisfiable and an unsatisfiable core φ_C is identified. Suppose that $\varphi_C = \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], [\mathit{CNF}(r_1 + r_2 \leq 1)], (x_1 \vee r_1, 100), (x_3 \vee r_2, 100), (x_2 \vee \bar{x}_1, 1), (\bar{x}_3 \vee x_1, 1)\}$. The formula is relaxed as in the original unsatisfiability-based algorithm. As a result, the formula is updated as

[1] Notice that initially we confirmed that the MaxSAT formula is not unsatisfiable due to the hard clauses. Since at each iteration at least one soft clause is relaxed, the working formula at some point becomes satisfiable.

follows:

$$\begin{aligned}\varphi_W = \quad & \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], \\ & (x_1 \vee r_1, 99), (x_3 \vee r_2, 99), (x_2 \vee \bar{x}_1 \vee r_3, 1), (\bar{x}_3 \vee x_1 \vee r_4, 1), \\ & (x_1 \vee r_1 \vee r_5, 1), (x_3 \vee r_2 \vee r_6, 1), \\ & [\mathit{CNF}(r_1 + r_2 \leq 1)], [\mathit{CNF}(r_3 + r_4 + r_5 + r_6 \leq 1)]\}\end{aligned} \tag{7.3}$$

The resulting formula is now satisfiable. Since there are no more partitions to be added to the working formula, we have found an optimal solution to the original weighted partial MaxSAT formula. Consider that the SAT solver returns the satisfying assignment $\nu = \langle \bar{x}_1, x_2, x_3, r_1, \bar{r}_2, \bar{r}_3, r_4, \bar{r}_5, \bar{r}_6 \rangle$*. In this example, the optimal solution has a value of a hundred and one, corresponding to sum of the weights of the unsatisfied soft clauses in the original formula.*

In Algorithm 5, a partition method must be used to split the set of soft clauses. Example 7.1 uses weights to split the soft clauses into two partitions. However, other forms of partitioning may be devised. In the next section we present different methods for the partitioning soft clauses in MaxSAT.

7.2 MaxSAT Partitioning

Different methods for partitioning soft clauses may be devised for MaxSAT. For example, for weighted partial MaxSAT we may use the weights of the soft clauses to build partitions. However, for partial MaxSAT problems weights cannot be used. Therefore, other methods need to be devised. In this section we present different methods for partitioning soft clauses for MaxSAT formulas, namely, weight-based partitioning [MML12d], hypergraph partitioning [MML12d] and community-based partitioning [MML13a]. The partition methods described in this section represent different implementations of the `partitionSoft` procedure in line 4 of Algorithm 5.

7.2.1 Weight-based Partitioning

For weighted partial MaxSAT, weight-based partitioning builds partitions where soft clauses with the same weight belong to the same partition. The motivation is that soft clauses with the same weight are more likely to be related to each other than to the remaining soft clauses. Next, we sort the partitions of soft clauses from the largest to the smallest weight, in order to try to find unsatisfiable cores with larger weights in the first iteration.

Next, we present a worst case example of what may happen if weight-based partitioning is not used.

Example 7.2. *Consider a weighted partial MaxSAT formula $\varphi = \varphi_h \cup \varphi_s$ such that:*

$$\begin{aligned} \varphi_h = &\quad \{[\bar{x}_1 \vee x_2], [x_1 \vee x_2], [x_3 \vee x_4], [\bar{x}_4]\} \\ \varphi_s = &\quad \{(\bar{x}_2 \vee x_4, 100), (\bar{x}_4 \vee x_3, 1)\} \end{aligned} \tag{7.4}$$

All clauses are given to the SAT solver. The formula is unsatisfiable and an unsatisfiable core φ_C is identified. Suppose that $\varphi_C = \{[\bar{x}_1 \vee x_2], [x_1 \vee x_2], [\bar{x}_4], (\bar{x}_2 \vee x_4, 100), (\bar{x}_4 \vee x_3, 1)\}$. Note that the clause $(\bar{x}_4 \vee x_3, 1)$ is not necessary to explain the unsatisfiability of φ, i.e φ_C will still be unsatisfiable without this clause. Nevertheless, it is not guaranteed that the SAT solver will return an unsatisfiable core without unnecessary clauses. The formula is relaxed as usual. The weight of $(\bar{x}_2 \vee x_4, 100)$ is decreased by 1 and a relaxed version of this clause is created. As a result, the formula is updated as follows:

$$\begin{aligned} \varphi_W = &\quad \{[\bar{x}_1 \vee x_2], [x_1 \vee x_2], [x_3 \vee x_4], [\bar{x}_4], \\ &\quad (\bar{x}_2 \vee x_4, 99), (\bar{x}_2 \vee x_4 \vee r_1, 1), (\bar{x}_4 \vee x_3 \vee r_2, 1), \\ &\quad [\mathit{CNF}(r_1 + r_2 \leq 1)]\} \end{aligned} \tag{7.5}$$

The resulting formula is still unsatisfiable. Consider the SAT solver has identified the core $\varphi_C = \{[\bar{x}_1 \vee x_2], [x_1 \vee x_2], [\bar{x}_4], (\bar{x}_2 \vee x_4, 99), (\bar{x}_4 \vee x_3 \vee r_2, 1)\}$. Note that we are again in the same situation as the previous iteration, i.e. when the formula is relaxed the weight of $(\bar{x}_2 \vee x_4, 99)$ is decreased by 1 and a relaxed version of this clause is created. In the worst case, this procedure can be repeated up to 100 times in order to completely relax this clause.

Now, consider the scenario where the soft clauses have been partitioned by weight. Therefore, we have two partitions $\gamma_1 = \{(\bar{x}_2 \vee x_4, 100)\}$ and $\gamma_2 = \{(\bar{x}_4 \vee x_3, 1)\}$. If we use Algorithm 5 with these two partitions, then we will obtain the following result. First, the hard clauses and the first partition of soft clauses are given to the SAT solver. This formula is unsatisfiable and the unsatisfiable core $\varphi_C = \{[\bar{x}_1 \vee x_2], [x_1 \vee x_2], [\bar{x}_4], (\bar{x}_2 \vee x_4, 100)\}$ is identified. The formula is relaxed and updated as follows.

$$\begin{aligned} \varphi_W = &\quad \{[\bar{x}_1 \vee x_2], [x_1 \vee x_2], [x_3 \vee x_4], [\bar{x}_4], \\ &\quad (\bar{x}_2 \vee x_4 \vee r_1, 100) \\ &\quad [\mathit{CNF}(r_1 \leq 1)]\} \end{aligned} \tag{7.6}$$

The resulting formula is satisfiable. Consider that the SAT solver returns the satisfying assignment $\nu = \langle \bar{x}_1, x_2, x_3, \bar{x}_4, r_1 \rangle$. *Even though the SAT solver returned a satisfying assignment, there are still partitions of soft clauses that are not yet added to the working formula. Therefore, the next partition* (γ_2) *is now added to* φ_W *and the resulting formula is given to the SAT solver. The resulting formula is still satisfiable. Consider that the SAT solvers returns the satisfying assignment* $\nu = \langle \bar{x}_1, x_2, x_3, \bar{x}_4, r_1 \rangle$. *Since there are no more partitions to be added to the working formula, we have found an optimal solution to the original weighted partial MaxSAT.*

Example 7.2 shows how weight-based partitioning can improve the original unsatisfiability-based algorithm. Even though this example presents the worst case scenario, it shows that partitioning by weight may reduce the number of iterations significantly since it reduces unnecessary duplications of soft clauses during the relaxation procedure. An important optimization when using weight-based partitioning is to dynamically put the soft clauses that are duplicated into the partition having soft clauses with the same weight. Note that this procedure may dynamically create new partitions.

Even though the proposed approach is novel, there is related work on using weights to guide the search. Lexicographical optimization [MSAGL11] is dedicated to solving problem instances where the optimality criterion is lexicographic. Given a sequence of cost functions, an optimization criterion is said to be lexicographic whenever there is a preference in the order in which the cost functions are optimized. The optimality criterion of a weighted partial MaxSAT formula is said to be lexicographic if it has multiple function costs, i.e. if it has a Boolean Multilevel Optimization [ALS09] criteria.

Definition 7.1 (Boolean Multilevel Optimization [ALS09])**.** *Consider we have a weighted partial MaxSAT formula* $\varphi = \varphi_h \cup \varphi_s$. *Moreover, consider the soft clauses have been partitioned by weight into n partitions ordered by descending weight,* $\gamma = \{\gamma_1 \cup \ldots \gamma_n\}$, *whereas* γ_1 *corresponds to the soft clauses with larger weight and* γ_n *to the soft clauses with smallest weight.*

φ *has a Boolean multilevel optimization criteria if and only if the following condition holds:*

$$\bigwedge_{i=1}^{n} (w_i > \sum_{i+1 \leq j \leq n} w_j \times |\gamma_j|), \tag{7.7}$$

where w_i *corresponds to the weight of soft clauses in partition* γ_i.

Example 7.3. *Consider the following weighted partial MaxSAT formulas* $\varphi = \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], (x_1, 100), (x_3, 100), (x_2 \vee \bar{x}_1, 1), (\bar{x}_3 \vee x_1, 1)\}$ *and* $\varphi' = \{[\bar{x}_2 \vee \bar{x}_1], [x_2 \vee \bar{x}_3], (x_1, 5), (x_3, 5), (x_2 \vee$

$\bar{x}_1, 3), (\bar{x}_3 \vee x_1, 2)\}$. *Moreover, consider that both* φ *and* φ' *have their soft clauses partitioned by weight.* φ *has the following two partitions:* $\gamma_1 = \{(x_1, 100), (x_3, 100)\}$, $\gamma_2 = \{(x_2 \vee \bar{x}_1, 1), (\bar{x}_3 \vee x_1, 1)\}$. φ' *has the following three partitions:* $\gamma'_1 = \{(x_1, 5), (x_3, 5)\}$, $\gamma'_2 = \{(x_2 \vee \bar{x}_1, 3)\}$, $\gamma'_3 = \{(\bar{x}_3 \vee x_1, 2)\}$.

The optimality criterion of φ *is lexicographic since it only has two partitions, and the partition* γ_1 *has weight 100, which is larger than* $1 \times |\gamma_2| = 2$. *On the other hand, the optimality criterion of* φ' *is not lexicographic since it has three partitions, and the partition* γ_1 *has weight 5, which is equal to* $3 \times |\gamma_2| + 2 \times |\gamma_1| = 5$.

When using lexicographical optimization to solve MaxSAT, soft clauses are grouped by their weight to iteratively find an optimal solution to each criterion. Therefore, our approach is equivalent to the lexicographical approach when the MaxSAT instance has a lexicographical optimization criterion. However, our approach is more general since it can be applied even if the MaxSAT instance does not have a lexicographical optimization criterion.

Independently, Carlos Ansótegui et al. [ABGL12] recently proposed a similar algorithm to the one presented in Algorithm 5. Their approach also considers the weights of soft clauses to find unsatisfiable cores with larger weights first. They present a stratified approach where soft clauses are ordered by descending weight. All soft clauses with weight larger than or equal to k are considered in the working formula. k is initially set to the largest weight of the soft clauses. When the SAT solver returns satisfiable, the value of k is updated to the largest weight of the soft clauses that are still not added to the working formula. This approach is equivalent to Algorithm 5 when using weight-based partitioning. However, for some problem instances, the weights of the soft clauses may not capture the structure of the formula. For example, if a formula has all soft clauses with different weights, then each partition has one soft clause. Due to this fact, it may be better to update k to a smaller value than the largest weight of the soft clauses. Carlos Ansótegui et al. [ABGL12] proposed a diversity heuristic that may update k to a value that is less than the largest weight of the soft clauses not yet added. This approach may add soft clauses with different weights in one iteration.

In contrast, our approach is based on partitioning. Therefore, when most soft clauses have distinct weights, we can use different partitioning methods. A possible alternative is graph-based partitioning. In the next sections we present alternative partitioning approaches based on the underlined problem structure.

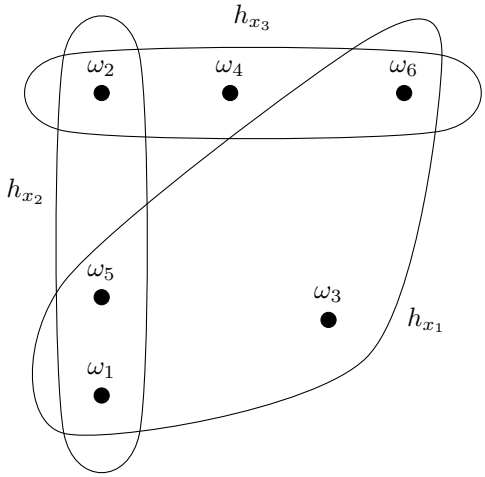


Figure 7.1: Hypergraph representation of the MaxSAT formula defined in Equation 7.1

7.2.2 Hypergraph Partitioning

A possible graph-based partitioning is hypergraph partitioning. A hypergraph is a generalization of a graph where an edge, also called hyperedge, can connect any number of vertices. A hypergraph representation of a weighted partial MaxSAT formula is built as follows. For each soft and hard clause there is a corresponding vertex in the hypergraph. Moreover, for each formula variable x_j there is an hyperedge connecting all vertices that represent soft or hard clauses containing variable x_j. This representation resembles the hypergraph obtained from a SAT formula [PG00].

Example 7.4. *Consider the weighted partial MaxSAT formula φ presented in equation 7.1. This formula has 6 clauses, $\varphi = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6\}$, where $\omega_1 = [\bar{x}_2 \vee \bar{x}_1]$, $\omega_2 = [x_2 \vee \bar{x}_3]$, $\omega_3 = (x_1, 100)$, $\omega_4 = (x_3, 100)$, $\omega_5 = (x_2 \vee \bar{x}_1, 1)$ and $\omega_6 = (\bar{x}_3 \vee x_1, 1)$.*

Figure 7.1 shows the hypergraph representation of φ. For each variable, x_1, x_2, x_3, there is a corresponding hyperedge, h_{x_1}, h_{x_2}, h_{x_3} connecting all vertices that represent clauses containing the corresponding variable.

After building the hypergraph, the tool HMETIS [KAKS99] is used as a black box to identify the partitions. In the experiments, HMETIS is configured to identify 16 partitions in each problem instance [MML12d]. Afterwards, for each partition only the soft clauses are considered. As a result, partitions containing only hard clauses are removed.

Example 7.5. *Consider the hypergraph presented in Figure 7.1 that corresponds to the weighted partial MaxSAT formula presented in equation 7.1. Assume that we give this hypergraph to the*

HMETIS *tool and try to partition the hypergraph into 2 disjoint partitions. The* HMETIS *tool will give the following partitions:* $\gamma_1 = \{\omega_3, \omega_6\}$, $\gamma_2 = \{\omega_1, \omega_2, \omega_4, \omega_5\}$. *After removing the hard clauses from each partition, we have:* $\gamma_1 = \{(x_1, 100), (\bar{x}_3 \vee x_1, 1)\}$, $\gamma_2 = \{(x_3, 100), (x_2 \vee \bar{x}_1, 1)\}$.

Note that the current hypergraph representation does not capture the weights of the soft clauses. Example 7.5 shows that the partitions given by HMETIS split soft clauses that have the same weight. This would not occur if one was using weight-based partitioning. Therefore, hypergraph partitioning is not expected to outperform weight-based partitioning when the weights capture the underline structure of the problem.

7.2.3 Community-based Partitioning

The identification of communities in SAT instances has been proposed in the past [AGCL12]. It has been shown to be effective in characterizing industrial SAT instances. For that, SAT instances are first represented as undirected weighted graphs and partitions of vertices (communities) are identified using a modularity measure. In this dissertation we use both graph representations described in [AGCL12] for SAT instances, namely the Variable Incidence Graph (VIG) and the Clause-Variable Incidence Graph (CVIG) model. In addition, a different weighting function is proposed.

Graph Representations

We start by defining an incidence function I on the formula variables x_j in the soft clauses φ_s as follows:

$$I(x_j) = 1 + \sum_{x_j \in \omega \ \wedge \ \omega \in \varphi_s} \frac{1}{|\omega|} \tag{7.8}$$

Notice that $I(x_j) = 1$ if variable x_j does not occur in any soft clause.

In the Variable Incidence Graph (VIG) model, a graph G is built such that for each variable x_j in the problem instance there is a corresponding vertex in G. Moreover, if x_j and x_k belong to the same clause (hard or soft), then there is an edge between the vertices corresponding to these variables with the following weight:

$$w(x_j, x_k) = \sum_{x_j, x_k \in \omega \ \wedge \ \omega \in \varphi} \frac{I(x_j) \cdot I(x_k)}{\begin{pmatrix} |\omega| \\ 2 \end{pmatrix}} \tag{7.9}$$

Observe that if we consider $I(x_j) = 1$ for all variables, then this weight function corresponds to the one proposed in [AGCL12], where all clauses are equally relevant. Note that a clause ω results into $\binom{|\omega|}{2}$ edges, one for each pair of variables. If all clauses have the same relevance, then the contribution of each clause to an edge is given by $\binom{|\omega|}{2}$. However, for MaxSAT one has to consider both soft and hard clauses. In our graph representation, more weight is given to clauses that establish edges between variables that occur in soft clauses. The motivation is to *fortify* the relationship between variables that occur in soft clauses.

Example 7.6. *Consider again the weighted partial MaxSAT formula φ presented in Equation 7.1. This formula has 6 clauses, $\varphi = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6\}$, where $\omega_1 = [\bar{x}_2 \vee \bar{x}_1]$, $\omega_2 = [x_2 \vee \bar{x}_3]$, $\omega_3 = (x_1, 100)$, $\omega_4 = (x_3, 100)$, $\omega_5 = (x_2 \vee \bar{x}_1, 1)$ and $\omega_6 = (\bar{x}_3 \vee x_1, 1)$.*

The incidence function I is defined as follows for variables x_1, x_2 and x_3.

$$\begin{aligned} I(x_1) &= 1 + 1 + 0.5 + 0.5 = 3 \\ I(x_2) &= 1 + 0.5 = 1.5 \\ I(x_3) &= 1 + 1 + 0.5 = 2.5 \end{aligned} \tag{7.10}$$

For each variable, we compute the contribution of each soft clause to the respective value of the incidence function I. For example, $I(x_1)$ is computed by summing the contributions of ω_3, ω_5, and ω_6, which are 1, 0.5, and 0.5, respectively.

Since all pairs of variables (x_i, x_j) appear in a clause together, we have a complete graph with 3 vertices and 3 edges. The weights of each edge are calculated using Equation 7.9 as follows.

$$\begin{aligned} w(x_1, x_2) &= \frac{3 \times 1.5}{\binom{2}{2}} + \frac{3 \times 1.5}{\binom{2}{2}} = 9 \\ w(x_1, x_3) &= \frac{3 \times 2.5}{\binom{2}{2}} = 7.5 \\ w(x_2, x_3) &= \frac{1.5 \times 2.5}{\binom{2}{2}} = 3.75 \end{aligned} \tag{7.11}$$

For each edge, we compute the contribution of each clause to the weight of the edge. For example, $w(x_1, x_2)$ is computed by summing the contributions of ω_1 and ω_5, both being of 4.5.

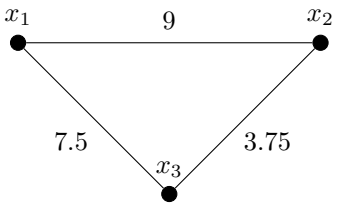


Figure 7.2: VIG representation of the MaxSAT formula defined in Equation 7.1

Figure 7.2 shows the weighted graph representation of the VIG model, where each vertex corresponds to a variable, and there is an edge between every two variables occurring in the same clause.

In the Clause-Variable Incidence Graph (CVIG) model, for each variable x_j and for each clause $\omega_i \in \varphi$, there is a corresponding vertex in graph G. In this model, edges only connect vertices representing a variable and a clause where that variable occurs. Hence, if a variable x_j occurs in clause ω_i, then there is an edge between those vertices with weight:

$$w(x_j, \omega_i) = \frac{I(x_j)}{|\omega_i|} \tag{7.12}$$

Example 7.7. *Consider again the weighted partial MaxSAT formula φ presented in equation 7.1. This formula has 6 clauses, $\varphi = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6\}$, where $\omega_1 = [\bar{x}_2 \vee \bar{x}_1]$, $\omega_2 = [x_2 \vee \bar{x}_3]$, $\omega_3 = (x_1, 100)$, $\omega_4 = (x_3, 100)$, $\omega_5 = (x_2 \vee \bar{x}_1, 1)$ and $\omega_6 = (\bar{x}_3 \vee x_1, 1)$.*

The incidence function I for x_1, x_2 and x_3 has the same values as the ones presented in Example 7.6. The weights of the edges are given by Equation 7.12 and are defined as follows.

$$\begin{aligned}
&w(x_1, \omega_1) = \tfrac{3}{2} = 1.5, \quad w(x_2, \omega_1) = \tfrac{1.5}{2} = 0.75, \quad w(x_3, \omega_2) = \tfrac{2.5}{2} = 1.25, \\
&w(x_1, \omega_3) = \tfrac{3}{1} = 3, \quad\;\; w(x_2, \omega_2) = \tfrac{1.5}{2} = 0.75, \quad w(x_3, \omega_4) = \tfrac{2.5}{1} = 2.5, \\
&w(x_1, \omega_5) = \tfrac{3}{2} = 1.5, \quad w(x_2, \omega_5) = \tfrac{1.5}{2} = 0.75 \quad\; w(x_3, \omega_6) = \tfrac{2.5}{2} = 1.25 \\
&w(x_1, \omega_6) = \tfrac{3}{2} = 1.5
\end{aligned} \tag{7.13}$$

Figure 7.3 shows the weighted graph representation of the CVIG model of φ. Vertices correspond to clauses, and variables and edges connect variables to the clauses they belong to.

Community Identification

After building a graph representation for the problem instance, we are interested in making explicit the hidden structure of the MaxSAT formula by identifying partitions in the graph. Clearly one can devise many different ways of partitioning. Therefore, it is necessary to evaluate the quality

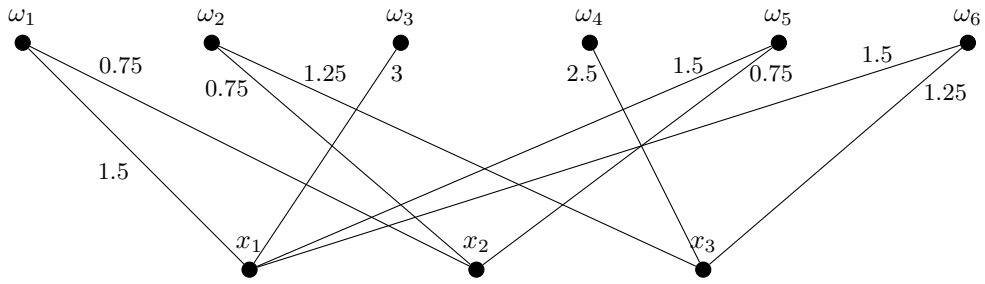


Figure 7.3: CVIG representation of the MaxSAT formula defined in Equation 7.1

of a given set of partitions.

In recent years, the use of modularity measures became common for the identification of communities in graphs [GN02a, NG04, RCC+04]. The main goal of the modularity measure is to evaluate the quality of the communities in a graph where vertices inside a community should be densely connected, while vertices assigned to different communities should be sparsely connected. Let $G = (V, w)$ denote a complete undirected weighted graph where V is the set of vertices and $w : V \times V \rightarrow \mathbb{R}$ is a weight function for each pair of vertices. If an edge does not occur in G, then it has weight 0. Let $C = \{C_1, C_2, \ldots, C_n\}$ denote a set of communities such that every vertex $u \in V$ is assigned to one and only one community in C. Hence, the modularity value Q of the set of communities C in graph G can be defined as follows [NG04]:

$$Q = \sum_{C_k \in C} \left[\frac{\sum\limits_{i,j \in C_k} w(i,j)}{m} - \left(\frac{\sum\limits_{i \in C_k} \sum\limits_{j \in V} w(i,j)}{2m} \right)^2 \right] \tag{7.14}$$

where $m = \sum_{i,j \in V} w(i,j)$ denotes the sum of the weights of all the edges in G.

One drawback of community identification using modularity measures is that finding a set of communities with an optimal modularity value is computationally hard [BDG+06]. As a result, several approximation algorithms have been proposed [CNM04, PL06a, BGLL08]. In this dissertation the method proposed in [BGLL08] is used.

From Communities to Partitions

After identifying the communities in the graph, one must build the set of partitions to be used in Algorithm 5. When using the CVIG model, building partitions of soft clauses is straightforward since clauses are directly represented in the graph. For each community with vertices representing soft clauses, there is a corresponding partition containing the respective soft clauses in the

community. After building the partitions, these are sorted by ascending size with respect to the number of soft clauses. Therefore, partitions with smaller size are considered first in Algorithm 5. This ordering allows the algorithm to search for unsatisfiable cores on the smallest possible subset of soft clauses.

In the VIG model, only variables are represented in the graph. Therefore, given the set of communities C, a soft clause ω belongs to the community C_k that maximizes $|C_k \cap \omega|$, i.e. C_k is the community with the largest number of variables in ω. In case of a tie, ω is assigned to the community of the lowest index variable in ω. After assigning all soft clauses to communities, partitions to be used in Algorithm 5 are built as in the CVIG model.

Example 7.8. *Consider the graphs given in Examples 7.6 and 7.7. Moreover, consider that these graphs are partitioned into communities using the modularity approach discussed previously. The partitioning of the VIG representation will result into a single partition:* $\gamma_1 = \{x_1, x_2, x_3\}$. *Since we only have one partition, this partition will contain all soft clauses. When this occurs, the performance of Algorithm 5 will be the same as the original unsatisfiability-based algorithm.*

On the other hand, the CVIG representation will result into two partitions: $\gamma'_1 = \{\omega_1, \omega_3, \omega_5, x_1, x_2\}$, $\gamma'_2 = \{\omega_2, \omega_4, \omega_6, x_3\}$. *After removing the hard clauses and the variables from the partitions, we are left with the following partitions:* $\gamma'_1 = \{(x_1, 100), (x_2 \vee \bar{x}_1, 1)\}$, $\gamma'_2 = \{(x_3, 100), (\bar{x}_3 \vee x_1, 1)\}$.

Similarly to the hypergraph representation, VIG and CVIG representations do not capture the weights of the soft clauses. Therefore, weight-based partitioning is expected to perform better than graph-based partitioning when weights capture the structure of the problem.

7.3 Experimental Results

This section evaluates the impact of partitioning in unsatisfiability-based algorithms when solving weighted partial MaxSAT and partial MaxSAT benchmarks. The partitioning techniques described in the previous section were implemented on top of WBO (version 2.0, 2012) [MMSP09, MML12d, MML13a]. For weighted partial MaxSAT, all experiments were run on instances from the crafted and industrial categories of the MaxSAT Evaluation of 2011[2]. Even though unsatisfiability-based algorithms perform better on industrial benchmarks, we also included crafted instances since there are only a small number of industrial instances. For partial MaxSAT, all experiments were run on instances from the industrial category. The cost of using the tool HMETIS and of building the

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

Table 7.1: Number of instances solved by each solver

Benchmark	#I	WBO	WEIGHT	HYP	VIG	CVIG	RDM	VBS
auc-paths	86	0	9	0	0	0	0	9
auc-scheduling	84	0	78	1	1	1	0	78
planning	56	34	51	34	35	36	24	51
warehouses	18	4	1	18	6	4	1	18
miplib	12	0	2	1	0	1	0	2
net	74	47	0	41	46	56	0	66
wcsp-dir	21	6	6	6	6	6	4	6
wcsp-log	21	6	6	5	6	5	4	6
pedigrees	100	72	80	22	32	38	20	83
timetabling	26	4	5	2	2	2	1	5
upgradeability	100	100	100	99	100	100	100	100
Total	598	273	338	229	234	249	154	424

modularity-based communities is considered in the running time of the respective approach. For the majority of the benchmarks this cost is negligible, i.e. it takes less than 1 second.

The evaluation was performed on two AMD Opteron 6276 processors (2.3 GHz) running Fedora 18 with a timeout of 1,800 seconds and a memory limit of 16 GB.

7.3.1 Evaluation on Weighted Partial MaxSAT Benchmarks

Table 7.1 compares the different partitioning solvers against the original WBO that does not use any partitioning techniques when solving weighted partial MaxSAT instances. The benchmarks `pedigrees`, `timetabling` and `upgradeability` correspond to instances from the industrial category, whereas the remainder benchmarks correspond to instances from the crafted category.

The solver using weight-based partitioning is denoted by WEIGHT, whereas HYP denotes the solver using hypergraph partitioning. VIG and CVIG correspond to the community-based partitioning using the VIG and the CVIG graph representations described in section 7.2.3, respectively. To assess the quality of the new partitions, we have also implemented a random partitioning technique where each soft clause is placed randomly in one of the partitions. We denote this solver by RDM. Similarly to the hypergraph partitioning, 16 partitions are used. The clauses are distributed uniformly among the partitions. In addition to the solvers previously mentioned, we have also included the Virtual Best Solver (VBS) between all solvers. The column VBS shows the number of instances that were solved by either WBO, WEIGHT HYP, VIG, CVIG or RDM.

Weight-based partitioning outperforms WBO showing that partitioning soft clauses by weight can significantly improve the performance of unsatisfiability-based algorithms. Weight-based par-

titioning is more efficient than WBO on benchmarks where weights capture the structure of the problem. Weights are said to capture the structure of the problem when several soft clauses share the same weight, and so can be clustered into partitions with the same weight, thus forming partitions with a large number of soft clauses. From the tested benchmarks, the weight-based partitioning approach did not capture the structure of the problem for the `warehouses` and `net` benchmarks. For these benchmarks, the use of weights for partitioning would create over 1,000 partitions, each of them containing on average slightly less than 2 soft clauses. For benchmarks where weights do not capture the structure of the problem, weight-based partitioning significantly deteriorates the performance of the solver. On the other hand, for the benchmarks where weights capture the structure of the problem, weight-based partitioning is more efficient than WBO since it is able to find the optimal solution while making less calls to the SAT solver. For instances where weights form natural partitions, WBO performs on average 684 unsatisfiable SAT calls, whereas WEIGHT only needs 225. Moreover, WEIGHT is also able to find smaller unsatisfiable cores than WBO. On average, WEIGHT finds unsatisfiable cores with 52 soft clauses, whereas unsatisfiable cores in WBO have 77 soft clauses. Weight-based partitioning is particularly effective for the `auc-scheduling` and `planning` instances. For the `auc-scheduling` instances, WBO performs on average 1,695 unsatisfiable SAT calls and finds on average unsatisfiable cores with 62 soft clauses. On the other hand, WEIGHT performs on average 166 unsatisfiable SAT calls and finds on average unsatisfiable cores with 14 soft clauses. For the `planning` instances, WBO performs on average 109 unsatisfiable SAT calls and finds on average unsatisfiable cores with 88 soft clauses. In contrast, WEIGHT performs on average 18 unsatisfiable SAT calls and finds on average unsatisfiable cores with 48 soft clauses. These significant reductions in the number of unsatisfiable SAT calls and on the size of the unsatisfiable cores explain the effectiveness of WEIGHT for these benchmarks.

Graph-based partitioning has an orthogonal performance with respect to weight-based partitioning. Graph-based partitioning improves the performance of the solver on benchmarks where weights do not capture the structure of the problem, and deteriorates the performance of the solver on benchmarks where weights capture the structure of the problem. For the `warehouses` benchmarks, HYP was able to solve all of the benchmarks. On the other hand, for the `net` benchmarks, CVIG is able to improve the performance of the solver since it solved 9 instances more than WBO. For the `warehouses` instances, HYP performs on average 153 unsatisfiable SAT calls and finds on average unsatisfiable cores with 31 soft clauses. In contrast, WBO performs on average 944 unsatisfiable SAT calls and finds on average unsatisfiable cores with 215 soft clauses. This

significant reduction on the number of unsatisfiable SAT calls and on the size of the unsatisfiable cores explains why HYP is able to solve all `warehouses` instances. For the `net` instances, CVIG performs on average 679 unsatisfiable SAT calls and finds on average unsatisfiable cores with 37 soft clauses. On the other hand, WBO performs on average 692 unsatisfiable SAT calls and finds on average unsatisfiable cores with 38 soft clauses. Even though the reduction on the number of unsatisfiable SAT calls and on the size of the unsatisfiable cores is not as significant as on other benchmarks, it helps towards solving more 9 instances.

Graph-based partitioning is much worse than weight-based partitioning when the soft clauses form natural partitions. This phenomenon can be observed in the `auc-scheduling`, `planning` and `pedigrees` benchmarks. When comparing the average number of soft clauses and the average number of partitions generated by WEIGHT and CVIG we can have a better understanding of why CVIG performs so poorly. The `scheduling` instances have on average 160 soft clauses. For these instances, WEIGHT creates on average 9 partitions per instance, whereas CVIG creates on average 51 partitions. The `planning` instances have on average 315 soft clauses. For these instances, WEIGHT creates on average 30 partitions, whereas CVIG creates on average 58 partitions. Therefore, one can observe that the number of soft clauses in these benchmarks is small. Moreover, since CVIG creates a large number of partitions, each of them contains a small number of soft clauses thus leading to a detrimental effect on the performance of the solver. This effect is even more visible for the `pedigrees` benchmarks. These instances have on average 11,205 soft clauses. WEIGHT creates 2 partitions for all of these instances, whereas CVIG creates on average 2,374 partitions. This large number of partitions leads to the deterioration of the performance of the solver observed in Table 7.1. Note that a similar effect to CVIG is observed for both VIG and HYP.

Randomly partitioning the soft clauses leads to a significant deterioration in the performance of the solver. This shows that even though partitioning may improve the performance of the solver, if it does not take into account the structure of the formula, then it will degrade the performance of the solver.

The VBS shows that different partitions methods may perform better for different benchmarks. For the `net` benchmarks, 66 instances can be solved if we consider all the different graph-based partitioning methods. Therefore, the VBS solves more 10 instances than the CVIG method. For the `net` benchmarks, HYP is able to solve 6 instances not solved by CVIG, whereas VIG is able to solve 4 instances not solved by either HYP or CVIG. When weights captures the structure of the problem, the WEIGHT method is clearly superior to the graph-based approaches. Nevertheless, for

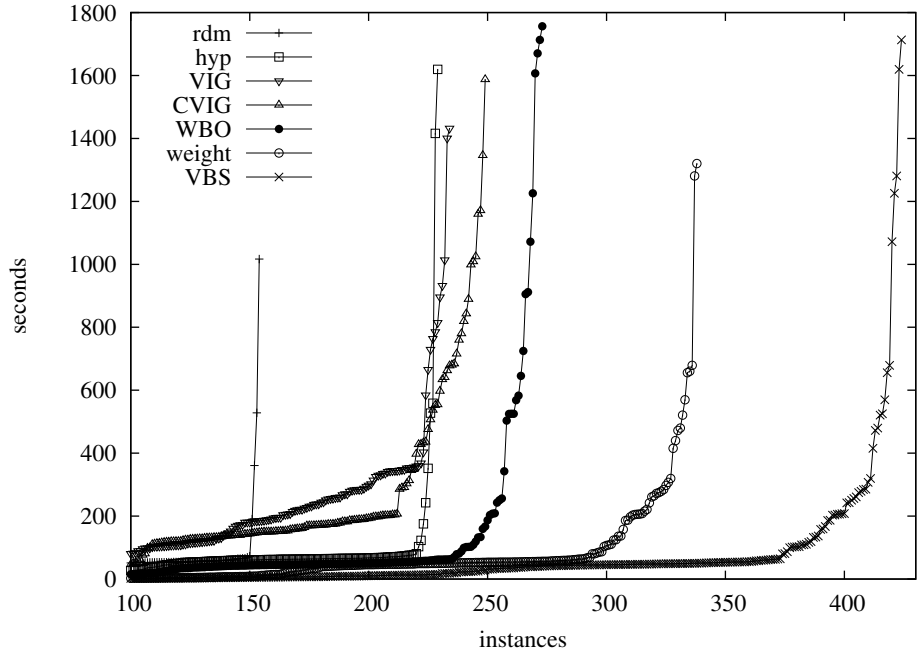


Figure 7.4: Comparison between different partitioning solvers

the `pedigrees` benchmarks, graph-based partitioning is able to solve 3 instances not solved by WEIGHT.

The values of VBS motivate the use of a dynamic heuristic that chooses between WEIGHT and one of the graph-based solvers. For example, one may use a dynamic heuristic [MML12d] that selects a graph-based partitioning solver when the number of partitions is large (> 300) and the average number of soft clauses in each partition is small (< 3). If we were using HYP as our graph-based solver, then we would solve 396 instances with this heuristic. On the other hand, if we were using CVIG as our graph-based solver, then we would solve 397 instances. With a dynamic heuristic one may easily solve more instances than with WEIGHT and close the gap to the VBS.

The cactus plot of Figure 7.4 results from the running times of the different solvers. The x-axis shows the number of solved instances, whereas the y-axis shows the running time in seconds. Randomly partitioning soft clauses is significantly worse than the remaining solvers. CVIG is the best performing solver among the solvers based on graph partitioning. However, all graph-based partitioning solvers exhibit similar performances. Since weights form natural partitions for the majority of the benchmarks, the overall performance of graph-based solvers is worse than our baseline solver WBO. On the other hand, this allows WEIGHT to significantly improve the performance of WBO. Moreover, if one considers the VBS between the different methods of partitioning,

Table 7.2: Number of instances solved by each solver

Benchmark	#I	WBO	HYP	VIG	CVIG	rdm	VBS
bcp-fir	59	40	26	28	29	18	46
bcp-hipp	55	21	15	21	20	15	24
bcp-msp	64	4	5	4	6	1	9
bcp-mtg	40	18	39	38	36	19	39
bcp-syn	74	32	31	35	32	30	38
circuit	4	1	2	2	2	2	2
haplotype	6	5	5	5	5	5	5
pbo-mqc	168	43	139	138	143	134	149
pbo-routing	15	15	6	15	6	3	15
protein	12	1	2	1	1	2	2
Total	497	180	269	287	280	229	329

then we can further increase the performance of WEIGHT.

7.3.2 Evaluation on Partial MaxSAT benchmarks

Table 7.2 compares the different partitioning solvers against the original WBO that does not use any partitioning techniques on partial MaxSAT instances. The solvers evaluated for partial MaxSAT are WBO, HYP, VIG, CVIG and RDM. Note that WEIGHT is not used for partial MaxSAT since all soft clauses have weight 1. If one uses WEIGHT, then one partition corresponding to the algorithm used in the WBO solver is obtained. In addition to the previously mentioned solvers, we have also included the VBS between all solvers.

Randomly partitioning the soft clauses can have a detrimental effect on the performance of the solver for several classes of benchmarks. However, it can also significantly improve the performance of the solver on other classes of benchmarks, such as `pbo-mqc`. Nevertheless, other partition methods based on structural information of the formula are clearly better.

Community-based partitioning outperforms hypergraph partitioning. Note that hypergraph partitioning creates a fixed number of partitions. However, in community-based partitioning, the number of partitions is dynamic and depends on the structure of the formula. This may explain the effectiveness of community-based partitioning.

Our motivation for partitioning is to identify easier to solve subproblems. As a side effect, this may lead to finding smaller unsatisfiable cores at each call of the SAT solver. On average, WBO finds unsatisfiable cores with 110 soft clauses, whereas unsatisfiable cores in VIG have 66 soft clauses. The other solvers using partitions also behave similarly and find on average smaller unsatisfiable cores than WBO. This behavior is particularly visible on the `pbo-mqc` benchmarks

For these benchmarks, WBO finds on average unsatisfiable cores with 167 soft clauses, whereas unsatisfiable cores in VIG have only 47 soft clauses.

However, if the partitions are not adequate, then we may split *related* soft clauses between different partitions. This may prevent the solver from finding small unsatisfiable cores. For example, this behavior is observed in the `routing` benchmarks. On average, WBO finds unsatisfiable cores with 7 soft clauses, whereas CVIG finds unsatisfiable cores with 45 soft clauses. Due to an inadequate partitioning, CVIG is only able to solve 6 out of 15 instances of `routing`. Note that VIG can solve all `routing` instances since the partitions used allowed to find on average unsatisfiable cores with 9 soft clauses. Example 7.9 shows the impact that inadequate partitioning may have on the performance of the solver.

Example 7.9. *Consider a partial MaxSAT formula $\varphi = \varphi_h \cup \varphi_s$. Assume that φ_h contains $[x_1 \vee x_2 \vee x_3]$, and that φ_s contains $(\bar{x}_1), (\bar{x}_2), (\bar{x}_3)$. If these soft clauses are placed in the same partition then we can find a trivial unsatisfiable core with the soft clauses $(\bar{x}_1), (\bar{x}_2), (\bar{x}_3)$ and the hard clause $[x_1 \vee x_2 \vee x_3]$. Now, consider the worst case scenario where $(\bar{x}_2)$, $(\bar{x}_3)$ and $(\bar{x}_4)$ are placed in different partitions with other soft clauses. Let us assume that we first try to find unsatisfiable cores in the partition that contains $(\bar{x}_2)$. In this case, we may find an unsatisfiable core containing $(\bar{x}_2)$ and other soft clauses. Note that each time a new unsatisfiable core φ_C is found, all soft clauses in φ_C are relaxed. Therefore, after several iterations, when the working formula finally contains $(\bar{x}_2)$, $(\bar{x}_3)$ and $(\bar{x}_4)$, we may have already relaxed these soft clauses several times. If this is the case, then we will no longer be able to find the small unsatisfiable core that could be identified if no partitioning was used.*

It was observed that the inadequate partitioning presented in Example 7.9 occurs frequently in some classes of benchmarks, such as `fir` and `routing`. Moreover, if a random partitioning is used, then this problem is even more accentuated. This may explain why random partitioning deteriorates the performance of the solver for several benchmark sets. On the other hand, this shows that an adequate partitioning of the formula is essential for the effectiveness of the solver.

The cactus plot of Figure 7.5 results from the running times of the different solvers. We can distinguish between three classes of solvers: (i) solvers that do not use partitioning (WBO), (ii) solvers that use random partitioning (RDM) and (iii) solvers that use the structure of the formula to create the partitions (HYP, CVIG and VIG). Even random partitioning improves the overall performance of the solver. However, when the structure of the formula is taken into account, the performance of the solver is further improved.

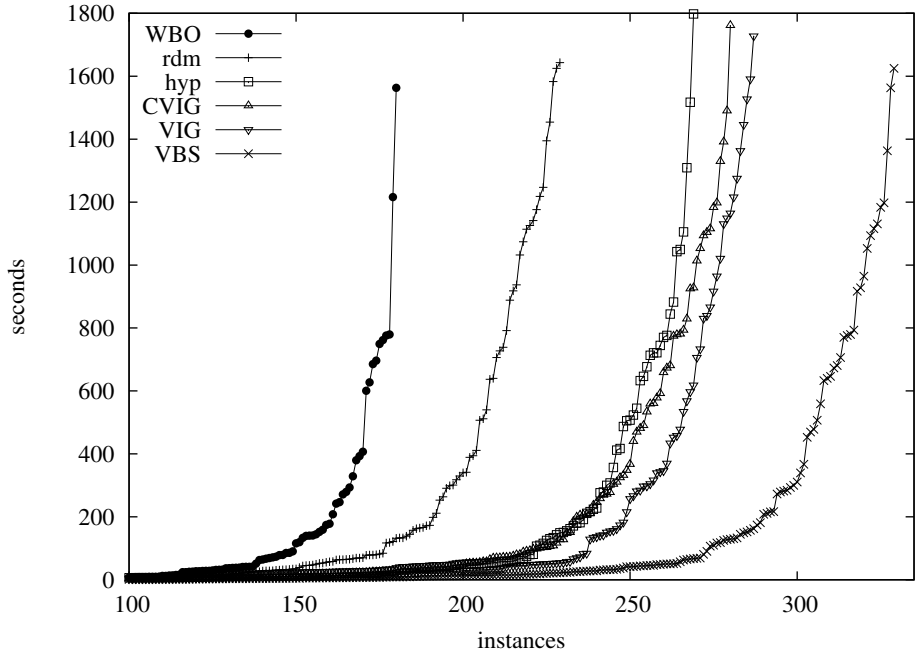


Figure 7.5: Comparison between different partitioning solvers

The VBS is able to solve 42 more instances than the VIG. Different graph-based partition methods may solve different instances thus increasing the number of instances solved by the VBS. Even though partitioning approaches outperform WBO on most benchmarks, there are some benchmarks where partitioning may degrade the performance of the solver. Therefore, there are some benchmarks where the best approach is the one that does not partition the soft clauses.

7.4 Summary

This chapter described how unsatisfiability-based algorithms can be enhanced by partitioning soft clauses. Experimental results have shown that partitioning the soft clauses can significantly improve the unsatisfiability-based algorithm WBO. For weighted partial MaxSAT benchmarks, weight-based partitioning has shown to be the most effective method of partitioning since for the majority of the instances the weights form natural partitions. However, when weights do not form natural partitions, weight-based partitioning is not adequate and therefore graph-based partitioning should be used. For these benchmarks, graph-based partitioning has shown to improve the performance of WBO. Randomly partitioning the soft clauses leads to the deterioration of the performance of the solver. This supports the idea that for solving weighted partial MaxSAT it is

particularly important to consider the structure of the formula when using a partitioning approach.

For partial MaxSAT benchmarks, weight-based partitioning cannot be used since all soft clauses have weight 1. However, for these benchmarks, graph-based partitioning has shown to significantly improve the performance of the solver. Even randomly partitioning the soft clauses leads to improvements on solving partial MaxSAT instances. However, if the structure of the formula is taken into consideration, then we can further improve the effectiveness of the solver. This supports the idea that using the structure of the formula to guide the search improves the performance of the solver and provides a strong stimulus for future research.

As future work, we propose to capture the weights of soft clauses on the graph-based representations. This should further increase their performance for solving weighted partial MaxSAT instances. Furthermore, note that the partitioning approaches proposed in this dissertation are not limited to the WBO algorithm and may be used in other unsatisfiability-based algorithms (e.g. [ABL09, HMMS11]).

8

Conclusions and Future Work

In the last years the number of cores and processors have been continuously increasing. As a result, an increasing number of parallel SAT solvers have come to light in the recent past. The use of SAT is widespread with many practical applications and it is clear that the optimization version of SAT, i.e. MaxSAT, can be applied to solve many real-world optimization problems. The competitive performance and robustness of MaxSAT solvers is certainly required to achieve this goal. As multicore architectures become predominant, it is crucial to develop new frameworks and techniques that take advantage of this new reality in order for MaxSAT solvers to continue evolving.

This dissertation presented several contributions on MaxSAT. First, we have proposed a dynamic encoding heuristic for cardinality constraints. Next, we have presented PWBO, the first parallel MaxSAT solver. We have also presented a deterministic version of PWBO that can be used for applications where results need to be reproducible. Using the deterministic version of PWBO we have performed a fair comparison between different clause sharing heuristics for parallel MaxSAT. Finally, we have also proposed partitioning techniques to enhance the performance of sequential unsatisfiability-based MaxSAT algorithms.

Next, we present a detailed description of the contributions of this dissertation and directions for future work.

8.1 Contributions

This sections presents a detailed description of the following contributions presented in this thesis: (i) dynamic encoding heuristic for cardinality constraints, (ii) parallel MaxSAT algorithms, (iii) deterministic parallel MaxSAT algorithms, (iv) clause sharing heuristics, and (v) partitioning algorithms for MaxSAT.

Dynamic Encoding Heuristic for Cardinality Constraints

This dissertation examined a large number of cardinality encodings and evaluated their performance for different MaxSAT algorithms. Moreover, a dynamic encoding heuristic for *at-most-k* cardinality constraints is presented. Experimental results have shown that the dynamic encoding heuristic outperforms all other encodings for linear search algorithms.

Parallel MaxSAT

This dissertation introduced PWBO, the first parallel solver for MaxSAT. Three versions of PWBO were proposed. The first version, PWBO-2, uses two threads, one thread searching on the lower bound value of the optimal solution, and another thread searching on the upper bound value of the optimal solution. The second version, PWBO-P, is based on a portfolio approach using several threads to simultaneously search on the lower and upper bound values of the optimal solution. These threads differ between themselves in the encoding used for cardinality constraints, thus increasing the diversification of the search. The third version, PWBO-S, is based on a splitting approach searching on different values of the upper bound. The parallel search on the local upper bound values leads to updates on the lower and upper bound values that reduce the search space.

Experimental results show that PWBO improves in performance with the increasing number of threads. PWBO-S-T4 outperforms PWBO-P-T4 on most instances. However, PWBO-S does not scale very well and its performance with 8 threads is only slightly better than with 4 threads. On the other hand, PWBO-P performance improves significantly when increasing the number of threads from 4 to 8. This shows that, even with 8 threads, using different cardinality encodings still increases the diversity of the search.

PWBO-T2 has been submitted to the MaxSAT evaluations of 2011 and 2012. The MaxSAT evaluation is an independent evaluation that aims at assessing the state of art in the field of MaxSAT solving. These MaxSAT evaluations were run on AMD Opteron 242 Processors (1.5 GHz) with a memory limit of 450 MB and a time limit of 1,800 seconds (CPU time). Note

that PWBO-T2 is run with 2 threads and therefore only 900 CPU seconds are allowed for each thread. Moreover, 450 MB is a small memory limit that may impose problems to our solver since each thread has its own clause database. Nevertheless, even with these limitations, PWBO-T2 placed second in the industrial category of partial MaxSAT (2011 and 2012). Moreover, a version of PWBO-T2 was also submitted to the weighted category of the MaxSAT evaluation of 2012. This version won the industrial category of weighted partial MaxSAT (2012). The results from the MaxSAT evaluations show that PWBO provides a valuable contribution to the research in MaxSAT.

Deterministic Parallel MaxSAT

Parallel MaxSAT solvers can improve the performance of sequential MaxSAT solvers. However, they cannot be used in application domains that require reproducible results. This dissertation presented the first deterministic parallel MaxSAT solver. Different approaches for thread synchronization were studied, namely, standard, period and dynamic synchronization. Dynamically changing the number of conflicts that each thread requires to reach a synchronization point improves the performance of the deterministic solver. Moreover, the dynamic synchronization also reduces the idle time of the solver. An analysis of the performance of our deterministic parallel MaxSAT solver with the non-deterministic versions shows that the non-deterministic solver can only solve a few more instances. Moreover, for instances that are solved by both solvers, the deterministic solver exhibits a performance that is similar to the non-deterministic version.

Clause Sharing Heuristics

Another application of deterministic solvers is to evaluate the performance of heuristics. In this case, deterministic solvers allow having a parallel experimental setup such that the only variation is the heuristic. In this dissertation we have evaluated different sharing heuristics for parallel MaxSAT solving. Moreover, a new heuristic based on the notion of freezing is proposed. This heuristic delays importing shared learned clauses by a given thread. These clauses are frozen until they are considered relevant in the context of the current search.

Experimental results show that sharing learned clauses in a portfolio-based parallel MaxSAT solver does not increase significantly the number of solved instances. However, it does allow a considerable reduction of the solving time. Moreover, the new freezing heuristic outperforms all other heuristics both in solving time and number of solved instances.

Improving Sequential MaxSAT

Partitioning the soft clauses has shown to significantly improve the unsatisfiability-based algorithm of WBO. For weighted partial MaxSAT benchmarks, weighted-based partitioning has shown to be the most effective method of partitioning. However, for partial MaxSAT benchmarks, weight-based partitioning is not adequate since all soft clauses have weight 1. For these benchmarks, we have proposed different graph-based partitioning methods, namely, hypergraph partitioning and community-based partitioning. Experimental results show that for both weighted partial MaxSAT and partial MaxSAT, partitioning the MaxSAT formula enabled the SAT solver to return significantly smaller unsatisfiable cores. This supports the idea that using the structure of the formula to guide the search improves the performance of the solver and provides a strong stimulus for future research.

8.2 Future Work

This dissertation presented the first parallel algorithms for MaxSAT. We took the first step towards using multicore architectures to improve the performance of MaxSAT solvers. In this process, we have also encountered several research opportunities that should be explored in the future. Next, we present two key issues that should be addressed: scalability and applications of partitioning.

Scalability

With the current increase in the number of cores, scalability becomes an important issue. In this dissertation, we have presented parallel algorithms for MaxSAT and shown that they perform well in a small number of cores. However, if one wants to use a larger number of cores or to explore distributed architectures, then we must improve the scalability of our approach.

Portfolio-based approaches are usually limited in terms of scalability since they are dependent of finding complementary approaches. Even though we have shown that the search can be diversified through the use of different cardinality encodings, we can further diversify the search if we consider more MaxSAT algorithms. Currently, we are using an unsatisfiability-based algorithm [MMSP09, ABL09] for our lower bound search and a linear search algorithm [BP10, AZFH12] for our upper bound search. However, we can extend our portfolio of algorithms by considering other kinds of unsatisfiability-based algorithms [ABL09, ABL10a, HMMS11] and linear search algorithms [AKFH11, MML13c]. This should further extend the diversity of our approach and may

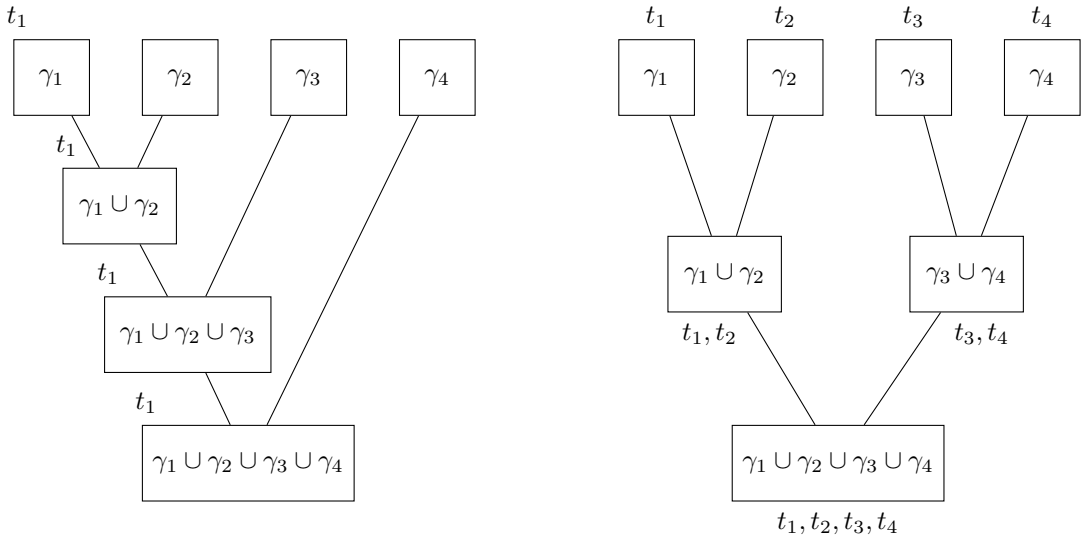


Figure 8.1: Sequential (left) and parallel (right) approaches based on partitioning

improve the performance of PWBO when using a larger number of cores.

Another research direction would be to build a hybrid version between the splitting and the portfolio approaches presented in this dissertation. We could start with a splitting strategy and when the interval between the lower and upper bound values becomes small change to a portfolio approach. A hybrid approach may be more suitable for a large number of cores since we may reduce the possible values of the optimal solution to a small interval. Switching to a portfolio approach with this knowledge may considerably improve the performance of our solver.

Finally, we can also increase the diversity of the search by exploring the parallelism given by other approaches. For example, the partitioning approach presented in Chapter 7 can be parallelized. In the next section we explore this idea and other applications of partitioning.

Applications of Partitioning

In this dissertation, we have shown that partitioning can greatly improve the performance of unsatisfiability-based algorithms. However, partitioning is not limited to improving sequential unsatisfiability-based algorithms as it can be extended to parallel approaches and to other kinds of sequential algorithms.

Figure 8.1 presents a sketch of a possible parallelization based on the partitioning approach. Consider a MaxSAT formula $\varphi = \varphi_h \cup \varphi_s$. Moreover, suppose that φ_s is split into 4 partitions, $\gamma_1, \gamma_2, \gamma_3, \gamma_4$. On the left, we present the sequential approach described in Chapter 7, which uses one thread (t_1). The algorithm based on partitioning starts by considering the first partition γ_1.

If there are no more unsatisfiable cores when considering γ_1, then γ_2 is added to the formula. This procedure is repeated until we have added all partitions to the formula and the formula becomes satisfiable.

On the right, we present a hybrid parallel approach based on partitioning and portfolio. Consider we have 4 threads, t_1, t_2, t_3, t_4. We start by assigning a partition to each thread. In parallel, each thread searches for unsatisfiable cores in its own partition. After all threads are idle, we can merge some of the partitions and start a new iteration. For example, we can merge γ_1 with γ_2 and use threads t_1 and t_2 to search for unsatisfiable cores in those partitions. Similarly, we can also merge γ_3 and γ_4 and use threads t_3 and t_4 to search for unsatisfiable cores in those partitions. In this iteration, we have a hybrid approach between partitioning and portfolio since more than one thread is searching in the same partition. To increase the diversity of the search, threads that search on the same partition should differ between themselves. For example, they may use different cardinality encodings as proposed in this dissertation. Finally, all partitions are merged. In this last iteration, all threads are searching on the same partition. Therefore, the last iteration corresponds to the portfolio approach presented in this dissertation. One of the drawbacks of this approach is load balancing. Some threads may be quickly idle where others are still finding unsatisfiable cores in their partitions. However, we may improve our first proposal by using a dynamic workload procedure. If one thread becomes idle, then it may help some other thread on finding unsatisfiable cores in its partition. Therefore, every time a thread becomes idle we may have a portfolio approach on partitions that still have threads looking for unsatisfiable cores.

Partitioning can also be used in other kinds of sequential MaxSAT algorithms. For example, one may use partitioning in linear search algorithms. As seen in Chapter 3, linear search algorithms start by adding a new relaxation variable to each soft clause and solving the resulting formula with a SAT solver. Whenever a model is found, a new constraint on the relaxation variables is added such that models with a greater or equal value are excluded. However, if the problem instance has a large number of relaxation variables, then adding a new constraint over these variables can lead to the exploration of a much larger search space. Instead of adding all relaxation variables to the new constraint, one may use a partitioning approach that starts with a subset of relaxation variables and incrementally adds new relaxation variables to the new constraint. We have explored this approach recently, and preliminary results show that partitioning may improve the performance of linear search algorithms [MML13c]. This provides a strong stimulus for further research on other forms of partitioning for MaxSAT algorithms.

Bibliography

[ABGL12] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-Based Weighted MaxSAT Solvers. In *Proc. International Conference on Principles and Practice of Constraint Programming*, pages 86–101. Springer, 2012.

[ABH+08] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. A Generalized Framework for Conflict Analysis. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 21–27. Springer, 2008.

[ABL09] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 427–440. Springer, 2009.

[ABL10a] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. A New Algorithm for Weighted Partial MaxSAT. In *Proc. AAAI Conference on Artificial Intelligence*, pages 3–8. AAAI Press, 2010.

[ABL+10b] Josep Argelich, Daniel Le Berre, Inês Lynce, Joao Marques-Silva, and Pascal Rapicault. Solving Linux Upgradeability Problems Using Boolean Optimization. In *Workshop on Logics for Component Configuration*, pages 11–22, 2010.

[ABL13] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013.

[AGCL12] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The Community Structure of SAT Formulas. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 410–423. Springer, 2012.

[AHJ+12] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting Clause Exchange in Parallel SAT Solving. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 200–213. Springer, 2012.

[AKFH11] Xuanye An, Miyuki Koshimura, Hiroshi Fujita, and Ryuzo Hasegawa. QMaxSAT version 0.3 & 0.4. In *Workshop on First-Order Theorem Proving*, pages 7–15, 2011.

[ALM07] Josep Argelich, Chu Min Li, and Felip Manyà. An Improved Exact Solver for Partial Max-SAT. In *Proc. of the International Conference on Nonconvex Programming: Local and Global Approaches*, pages 230–231, 2007.

[ALMS11] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. On Freezing and Reactivating Learnt Clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 188–200. Springer, 2011.

[ALS09] Josep Argelich, Inês Lynce, and Joao P. Marques Silva. On Solving Boolean Multilevel Optimization Problems. In *Proc. International Joint Conferences on Artificial Intelligence*, pages 393–398. IJCAI/AAAI Press, 2009.

[AM04] Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables into problems with boolean variables. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 1–15. Springer, 2004.

[AN12] Roberto Asín and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, pages 1–21, 2012.

[ANORC10] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Practical Algorithms for Unsatisfiability Proof and Core Generation in SAT solvers. *AI Communications*, 23(2-3):145–157, 2010.

[ANORC11] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality Networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[AS09] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proc. International Joint Conferences on Artificial Intelligence*, pages 399–404. IJCAI/AAAI Press, 2009.

[AZFH12] Xuanye An, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. QMaxSAT: A Partial Max-SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:95–100, 2012.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In *Proc. International Conference on Principles and Practice of Constraint Programming*, pages 108–122. Springer, 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proc. Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer, 1999.

[BDG+06] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Goerke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. arXiv: physics, 0608255. 2006.

[BGLL08] Vicent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics*, 2008(10):P10008, 2008.

[Bie10] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. In *SAT Race, Solver Description*, 2010.

[Blo05] Wolfgang Blochinger. Towards Robustness in Parallel SAT Solving. In *International Conference on Parallel Computing*, pages 301–308, 2005.

[BMS00] Luís Baptista and Joao Marques-Silva. Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability. In *Proc. International Conference on Principles and Practice of Constraint Programming*, pages 489–494. Springer, 2000.

[BP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[BR05] Markus Büttner and Jussi Rintanen. Satisfiability Planning with Constraints on the Number of Actions. In *International Conference on Automated Planning and Scheduling*, pages 292–299. AAAI Press, 2005.

[BS96] Max Böhm and Ewald Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.

[BS97] Roberto J. Bayardo and Robert C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proc. AAAI Conference on Artificial Intelligence*, pages 203–208. AAAI Press, 1997.

[BSK03] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning. *Journal of Parallel Computing*, 29(7):969–994, 2003.

[CBH+07] Fabien Corblin, Lucas Bordeaux, Youssef Hamadi, Eric Fanchon, and Laurent Trilling. A SAT-based approach to decipher Gene Regulatory Networks. In *Integrative Post-Genomics Conference*, 2007.

[Che10] Jingchao Chen. A New SAT Encoding of the At-Most-One Constraint. In *International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2010.

[CNM04] Aaron Clauset, Mark E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):066111, 2004.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proc. ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[Cou96] Olivier Coudert. On solving covering problems. In *Proc. Design Automation Conference*, pages 197–202. IEEE Computer Society Press, 1996.

[CS08] Geoffrey Chu and Peter J. Stuckey. PMiniSat - A parallelization of MiniSat 2.0. In *SAT Race 2008, Solver Description*, 2008.

[CSMSV10] Yibin Chen, Sean Safarpour, Joao Marques-Silva, and Andreas G. Veneris. Automated Design Debugging With Maximum Satisfiability. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 29(11):1804–1817, 2010.

[CZI10] Michael Codish and Moshe Zazon-Ivry. Pairwise Cardinality Networks. In *Proc. International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 154–172. Springer, 2010.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[DVSK09] Gilles Dequen, Pascal Vander-Swalmen, and Michaël Krajecki. Toward Easy Parallel SAT Solving. In *Proc. International Conference on Tools with Artificial Intelligence*, pages 425–432. IEEE Computer Society Press, 2009.

[ES03] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518. Springer, 2003.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[FG10] Alan M. Frisch and Paul A. Giannaros. SAT Encodings for the At-Most-k Constraint. In *International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2010.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 252–265. Springer, 2006.

[FPDN05] Alan M. Frisch, Timothy J. Peugniez, Anthony J. Doggett, and Peter Nightingale. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings. *Journal of Automated Reasoning*, 35(1-3):143–179, 2005.

[FS02] Sean Forman and Alberto Segre. NAGSAT: A Randomized, Complete, Parallel Solver for 3-SAT. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 236–243. Springer, 2002.

[GHJS10] Long Guo, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Diversification and Intensification in Parallel SAT Solving. In *Proc. International Conference on Principles and Practice of Constraint Programming*, pages 252–265. Springer, 2010.

[GLMSO10] Ana Graça, Inês Lynce, Joao Marques-Silva, and Arlindo L. Oliveira. Efficient and Accurate Haplotype Inference by Combining Parsimony and Pedigree Information. In *Algebraic and Numeric Biology*, pages 38–56. Springer, 2010.

[GN02a] Michelle Girvan and Mark E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821–7826, 2002.

[GN02b] Evgueni Goldberg and Yakov Novikov. BerkMin: a Fast and Robust SAT-Solver. In *Proc. International Conference on Design, Automation, and Test in Europe*, pages 142–149. IEEE Computer Society Press, 2002.

[GN04] Ian P. Gent and Peter Nightingale. A new encoding of All Different into SAT. In *International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2004.

[GSK98] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting Combinatorial Search Through Randomization. In *Proc. AAAI Conference on Artificial Intelligence*, pages 431–437. AAAI Press, 1998.

[HBPG08] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause Learning Can Effectively P-Simulate General Propositional Resolution. In *Proc. AAAI Conference on Artificial Intelligence*, pages 283–290. AAAI Press, 2008.

[HJPS11] Youssef Hamadi, Saïd Jabbour, Cédric Piette, and Lakhdar Sais. Deterministic Parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.

[HJS09a] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-Based Clause Sharing in Parallel SAT Solving. In *Proc. International Joint Conferences on Artificial Intelligence*, pages 499–504. IJCAI/AAAI Press, 2009.

[HJS09b] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

[HKWB11] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer, 2011.

[HLO08] Federico Heras, Javier Larrosa, and Albert Oliveras. MiniMaxSAT: An efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.

[HMMS11] Federico Heras, António Morgado, and Joao Marques-Silva. Core-Guided Binary Search Algorithms for Maximum Satisfiability. In *Proc. AAAI Conference on Artificial Intelligence*, pages 36–41. AAAI Press, 2011.

[HMSW11] Youssef Hamadi, Joao Marques-Silva, and Christoph M. Wintersteiger. Lazy Decomposition for Distributed Decision Procedures. In *International Workshop on Parallel and Distributed Methods in Verification*, pages 43–54, 2011.

[HW12] Antti E. J. Hyvarinen and Christoph M. Wintersteiger. Approaches for Multi-Core Propagation in Clause Learning Satisfiability Solvers. Technical Report MSR-TR-2012-47, 2012.

[JM11] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proc. Conference on Programming Language Design and Implementation*, pages 437–446. ACM Press, 2011.

[KAKS99] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *IEEE Transactions on VLSI Systems*, volume 7, pages 69–79. IEEE Press, 1999.

[Kas90] Simon Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3):275–286, 1990.

[KK07] Will Klieber and Gihwon Kwon. Efficient CNF Encoding for Selecting 1 from N Objects. In *International Workshop on Constraints in Formal Verification*, 2007.

[KK11] Stephan Kottler and Michael Kaufmann. SArTagnan – A Parallel Portfolio SAT Solver with Lockless Physical Clause Sharing. In *Pragmatics of SAT Workshop*, 2011.

[KS92] Henry A. Kautz and Bart Selman. Planning as Satisfiability. In *Proc. European Conference on Artificial Intelligence*, pages 359–363. IOS Press, 1992.

[LHG08] Javier Larrosa, Federico Heras, and Simon de Givry. A logical approach to efficient Max-SAT solving. *Artificial Intelligence*, 172(2-3):204–233, 2008.

[LMP07] Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

[LS07] Han Lin and Kaile Su. Exploiting inference rules to compute lower bounds for MAX-SAT solving. In *Proc. International Joint Conferences on Artificial Intelligence*, pages 2334–2339. IJCAI/AAAI Press, 2007.

[LSB07] Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded SAT Solving. In *Proc. Asia and South Pacific Design Automation Conference*, pages 926–931. IEEE Computer Society Press, 2007.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Journal of Information Processing Letters*, 47(4):173–180, 1993.

[Man11] Norbert Manthey. Parallel SAT Solving - Using More Cores. *Pragmatics of SAT Workshop*, 2011.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proc. Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[MBC+06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *Proc. International Conference on Automated Software Engineering*, pages 199–208. IEEE Computer Society Press, 2006.

[MML10a] Vasco Manquinho, Ruben Martins, and Inês Lynce. Improving Unsatisfiability-Based Algorithms for Boolean Optimization. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 181–193. Springer, 2010.

[MML10b] Ruben Martins, Vasco Manquinho, and Inês Lynce. Improving Search Space Splitting for Parallel SAT Solving. In *Proc. International Conference on Tools with Artificial Intelligence*, pages 336–343. IEEE Computer Society Press, 2010.

[MML11a] Ruben Martins, Vasco Manquinho, and Inês Lynce. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *Proc. International Conference on Tools with Artificial Intelligence*, pages 313–320. IEEE Computer Society Press, 2011.

[MML11b] Ruben Martins, Vasco Manquinho, and Inês Lynce. Parallel Search for Boolean Optimization. In *RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion*, 2011.

[MML12a] Ruben Martins, Vasco Manquinho, and Inês Lynce. An Overview of Parallel SAT Solving. *Constraints*, 17(3):304–347, 2012.

[MML12b] Ruben Martins, Vasco Manquinho, and Inês Lynce. Clause Sharing in Deterministic Parallel Maximum Satisfiability. In *RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion*, 2012.

[MML12c] Ruben Martins, Vasco Manquinho, and Inês Lynce. Clause Sharing in Parallel MaxSAT. In *Proc. Learning and Intelligent Optimization Conference*, pages 455–460. Springer, 2012.

[MML12d] Ruben Martins, Vasco Manquinho, and Inês Lynce. On Partitioning for Maximum Satisfiability. In *Proc. European Conference on Artificial Intelligence*, pages 913–914. IOS Press, 2012.

[MML12e] Ruben Martins, Vasco Manquinho, and Inês Lynce. Parallel Search for Maximum Satisfiability. *AI Communications*, 25(2):75–95, 2012.

[MML13a] Ruben Martins, Vasco Manquinho, and Inês Lynce. Community-based Partitioning for MaxSAT Solving. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 182–191. Springer, 2013.

[MML13b] Ruben Martins, Vasco Manquinho, and Inês Lynce. Deterministic Parallel MaxSAT Solving (Under Review). *AI Communications*, 2013.

[MML13c] Ruben Martins, Vasco Manquinho, and Inês Lynce. Model-based Partitioning for MaxSAT Solving. In *RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion*, 2013.

[MMSP09] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for Weighted Boolean Optimization. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 495–508. Springer, 2009.

[MMZ+01] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535. ACM, 2001.

[MSAGL11] Joao Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.

[MSL07] Joao Marques-Silva and Inês Lynce. Towards Robust CNF Encodings of Cardinality Constraints. In *Proc. International Conference on Principles and Practice of Constraint Programming*, pages 483–497. Springer, 2007.

[MSLM09] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, volume 185, pages 131–153. IOS Press, 2009.

[MSS96] Joao Marques-Silva and Karem Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *Proc. International Conference on Computer-Aided Design*, pages 220–227. IEEE Computer Society Press, 1996.

[MSS99] Joao Marques-Silva and Karem Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[NG04] Mark E. J. Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(026113), 2004.

[Pap94] Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 294–299. Springer, 2007.

[PD11] Knot Pipatsrisawat and Adnan Darwiche. On the Power of Clause-Learning SAT Solvers as Resolution Engines. *Artificial Intelligence*, 175:512–525, 2011.

[PG00] Tai Joon Park and Allen Van Gelder. Partitioning Methods for Satisfiability Testing on Large Formulas. *Information and Computation*, 162(1-2):179–184, 2000.

[PL06a] Pascal Pons and Matthieu Latapy. Computing Communities in Large Networks Using Random Walks. *Journal of Graph Algorithms and Applications*, 10(2):191–218, 2006.

[PL06b] Steven Prestwich and Inês Lynce. Local Search for Unsatisfiability. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 283–296. Springer, 2006.

[Pre07] Steven Prestwich. Variable Dependency in Local Search: Prevention is Better than Cure. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pages 107–120. Springer, 2007.

[RCC+04] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101(9):2658–2663, 2004.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[SB10] Sven Schulz and Wolfgang Blochinger. Parallel SAT Solving on Peer-to-Peer Desktop Grids. *Journal of Grid Computing*, 8:443–471, 2010.

[Sin05] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proc. International Conference on Principles and Practice of Constraint Programming*, pages 827–831. Springer, 2005.

[SKC96] Bart Selman, Henry A. Kautz, and Bram Cohen. Local Search Strategies for Satisfiability Testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, pages 521–532. American Mathematical Society, 1996.

[SM08] Daniel Singer and Anthony Monnet. JaCk-SAT: A New Parallel Scheme to Solve the Satisfiability Problem (SAT) Based on Join-and-Check. In *Proc. Parallel Processing and Applied Mathematics*, pages 249–258. Springer, 2008.

[VSDK09] Pascal Vander-Swalmen, Gilles Dequen, and Michaël Krajecki. A Collaborative Approach for Multi-Threaded SAT Solving. *International Journal of Parallel Programming*, 37(3):324–342, 2009.

[ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[ZM03] Lintao Zhang and Sharad Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Proc. International Conference on Design, Automation, and Test in Europe*, pages 10880–10885. IEEE Computer Society Press, 2003.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *Proc. International Conference on Computer-Aided Design*, pages 279–285. IEEE Computer Society Press, 2001.