Universitat de Lleida

Encodings and Benchmarks for MaxSAT Solving

Alba Cabiscol Teixidó

Universitat de Lleida
Departament d'Informàtica i Enginyeria Industrial

TESI DOCTORAL

Encodings and Benchmarks for MaxSAT Solving

Memòria de treball presentada per n'**Alba Cabiscol Teixidó** a la *Universitat de Lleida* per a l'obtenció del títol de *Doctora per la Universitat de Lleida*. El treball contingut en aquesta memòria ha estat realitzat sota la direcció dels Dr. **Ramón Béjar Torres** i Dr. **Felip Manyà Serres**.

Lleida, juny de 2012

Abstract

Problem solving based on the Propositional Satisfiability Problem (SAT) is an active research area in Artificial Intelligence, and has been successfully applied to solve both academic and industrial decision problems. The success of SAT-based problem solving has in turn contributed to explore extensions of SAT such as Satisfiability Modulo Theories, Quantified Boolean Formulas, Many-Valued Satisfiability, Pseudo-Boolean Optimization, and Maximum Satisfiability.

In this thesis we focus on the Maximum Satisfiability Problem (MaxSAT), which is an optimization variant of SAT. Given a CNF formula ϕ, MaxSAT consists in finding a truth assignment that satisfies the maximum number of clauses of ϕ. Usually, we focus on the MaxSAT extension that associates weights with clauses, and where each clause is declared to be either soft or hard. In this case, known as Weighted Partial MaxSAT, an optimal solution is a truth assignment that satisfies all the hard clauses, and maximizes the sum of the weights of the satisfied soft clauses. Weighted Partial MaxSAT is called Weighted MaxSAT when all the clauses are soft, and Partial MaxSAT when all the soft clauses have the same weight.

Given the recent and promising results on MaxSAT, the main objective of this thesis is to contribute to develop appropriate MaxSAT technology for solving challenging NP-hard optimization problems by first reducing them to a MaxSAT formalism, and then finding a solution with a state-of-the-art MaxSAT solver. More specifically, our goal is twofold: firstly, improve the modeling of decision and optimization problems by defining original and efficient encodings from the Constraint Satisfaction Problem (CSP) into SAT, and extending them to map the Maximum Constraint Satisfaction

Problem (MaxCSP) into MaxSAT; and secondly, create MaxSAT instances generators of adjustable hardness to help identify potential enhancements and weaknesses of MaxSAT solvers, and assess the impact of individual and combined solving techniques.

Concerning encodings from CSP into SAT, we present two new encodings from CSP into SAT: the minimal support encoding and the interval-based support encoding. The minimal support encoding reduces the size of the support encoding, and the interval-based support encoding is the first support encoding containing only regular literals in which the size of the derived encoding does not grow exponentially in the worst case.

Concerning encodings from MaxCSP into Partial MaxSAT, we define and analyze a number of novel encodings that extend variants of the direct and support encodings from CSP into SAT. We identify the clauses that must be declared as hard and the clauses that must be declared as soft, and determine whether it is necessary to introduce auxiliary variables for producing encodings in such a way that the minimum number of falsified clauses in the generated Partial MaxSAT encoding is the same as the minimum number of violated constraints in the encoded MaxCSP instance.

Concerning the creation of MaxSAT instance generators of adjustable hardness, we describe generators that encode into MaxSAT the following combinatorial optimization problems: Max1+pSAT, Partial Max2SAT, MaxCut, and rectangular bin-packing.

The conducted empirical investigation provides evidence that the new encodings from CSP into SAT are particularly good for SAT solvers with conflict clause learning, the proposed encodings from MaxCSP into MaxSAT are well-suited for modelling optimization problems, and the created generators produce challenging benchmarks for MaxSAT solvers.

Acknowledgments

I would like to acknowledge the people who in some way have contributed to the success of this work. First of all, I would like to express my sincere gratitude to my supervisors, Ramón Béjar and Felip Manyà, whose support, encouragement and guidance during this time enabled me to develop this thesis.

I would also like to express my gratitude to all the members of the Artificial Intelligence Research Group of the Universitat de Lleida: Tere Alsinet, Carlos Ansótegui, Josep Argelich, Cèsar Fernández, Carles Mateu and Jordi Planes; the secretary of the department, Montse Espuñes; and the members of Escola Politècnica Superior with whom I shared friendship.

The work presented in this thesis was partially supported by the Generalitat de Catalunya under grant 2009-SGR-1434, and the Ministerio de Economía y Competitividad research projects CONSOLIDER CSD2007-0022, INGENIO 2010, TIN2009-14704-C03-01, TIN2010-20967-C04-01/03, and INNPACTO IPT-2011-1496-310000 (funded by the Ministerio de Ciencia e Innovación until 2011).

Finally, I would specially thank to my family for their constant support and help. This thesis is dedicated to all of them: my parents, Jaume and Maria.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and objectives

Solving combinatorial decision problems by first reducing them to the Propositional Satisfiability Problem (SAT), and then finding a solution with a state-of-the-art SAT solver, is considered to be a powerful generic problem solving approach. It has proven to be highly competitive in a variety of domains, including hardware verification [eSSMS99, MMZ+01, VB01, BK02, KSHK07], bioinformatics [LMS06b, LMS06a], planning [KS96, Kau06], and scheduling [BM00, ZLS04].

The success of SAT-based problem solving has in turn contributed to explore extensions and variants of SAT such as Satisfiability Modulo Theories (SMT) [BSST09], Quantified Boolean Formulas (QBF) [BB09, GMN09], Many-Valued Satisfiability (Many-Valued SAT) [BMC+07, BHM00], Pseudo-Boolean Optimization (PBO) [RM09], and Maximum Satisfiability (MaxSAT) [LM09]. Nowadays, the theoretical and empirical study of all the mentioned formalisms is an active research line in Artificial Intelligence (AI).

New logical and complexity results, novel solving techniques, highly optimized implementations, comprehensive benchmark libraries, and solver competitions bear witness of the interest and efforts that the AI community has devoted, over the last decade, to the formalisms that originated in the area of Satisfiability Testing.

The first assumption of this thesis is that any generic problem solving approach

consists of two interrelated components: the modeling component and the solving component, and in practice it is decisive to devise both good models and fast solvers. Nevertheless, in the Satisfiability Testing community, most efforts have focused on designing and implementing fast solvers, and the definition of efficient and effective encodings has not yet evolved as much as the development of solvers, despite of the fact that encodings have a significant impact on performance.

The second assumption of this thesis is that a decisive aspect for designing and implementing fast solvers is to have access to a suitable benchmark test suite. Benchmarking helps designers and developers identify potential enhancements, determine under which circumstances the solvers underperform, and assess the impact of individual and combined solving techniques. Ideally, the benchmark test suite should contain instances which are easy to generate, of adjustable hardness, and of as much diverse problems as possible [Mat09].

The third assumption of this thesis is that MaxSAT-based problem solving is becoming a competitive alternative for computing optimal solutions in combinatorial optimization problems. Contemporary MaxSAT solvers incorporate advanced and powerful solving techniques, and are able to deal with the variants of MaxSAT that are best suited for representing and solving problems with soft and hard constraints, as well as problems with preferences among the constraints [LM09].

Given the recent and promising results on MaxSAT, the main objective of this thesis is to contribute to develop appropriate MaxSAT technology for solving challenging combinatorial optimization problems by defining efficient and effective MaxSAT encodings, and creating MaxSAT instance generators of tunable hardness. More precisely, the specific objectives can be summarized as follows:

- Study the most popular encodings from the Constraint Satisfaction Problem (CSP) into SAT, and define original encodings that both reduce the space complexity and show a better performance profile on relevant problem classes. The focus on CSP is due to the fact that it provides a suitable framework for representing and solving combinatorial decision problem.

- Study how existing and new encodings from CSP into SAT can be extended to encode the Maximum Constraint Satisfaction Problem (MaxCSP) into MaxSAT, and identify which encodings perform better on representative MaxSAT solvers. Since MaxCSP provides a suitable framework for representing and solving combinatorial optimization problems, we aim at improving the modeling component of MaxSAT-based problem solving.

- Design and implement original MaxSAT instance generators of adjustable hardness of both random and structured instances, evaluate the generators on representative MaxSAT solvers, and identify under which circumstances the generated instances become harder.

1.2 Contributions

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

- Our first original contribution is the definition of two new encodings from CSP into SAT: the minimal support encoding and the interval-based support encoding. The minimal support encoding reduces the size of the support encoding, and the interval-based support encoding is the first support encoding containing only regular literals in which the size of the derived encoding does not grow exponentially in the worst case.

- Our second contribution is the study of encodings from MaxCSP into Partial MaxSAT that extend the existing encodings from CSP into SAT. We focus our attention on the variants of the direct and support encodings that use monosigned and/or regular literals, identify the clauses that must be declared as hard and the clauses that must be declared as soft, and determine whether it is necessary to introduce auxiliary variables for producing encodings in such a way that the minimum number of falsified clauses in the generated Partial MaxSAT encoding is the same as the minimum number of violated constraints.

- Our third contribution is an empirical analysis of the existing and new encodings from CSP into SAT, and from MaxCSP into Partial MaxSAT. The obtained results provide empirical evidence of the good performance profile of the new encodings on SAT solvers such as MiniSat and PrecoSAT that incorporate conflict-clause learning, as well as of the good performance profile when they are used to encode MaxCSP into Partial MaxSAT, and are solved with MaxSAT solvers such as MSUnCore [MSP08], SAT4J-Maxsat [LP10] and WMaxSatz [LMMP10, LMP07a].

- Our fourth contribution is the design and implementation of MaxSAT instance generators of tunable hardness. The generators encode into MaxSAT the following combinatorial optimization problems: Max1+pSAT, Partial Max2SAT, MaxCut, and bidimensional bin-packing. The conducted experimentation provides empirical evidence that the created generators produce challenging benchmarks for MaxSAT solvers.

1.3 Publications

Some of the results presented in this thesis have already been published in journals and conference proceedings. The list of publications, in chronological order, is the following one:

– Ramón Béjar, Felip Manyà, Alba Cabiscol, Cèsar Férnandez, and Carla P. Gomes. A Many-Valued Approach to Solving Combinatorial Problems. *Discrete Applied Mathematics*, 155 (12): pages 1613–1626, 2007.

– Josep Argelich, Alba Cabiscol, Inês Lynce and Felip Manyà. Encoding MaxCSP into Partial MaxSAT. In *38th International Symposium on Multiple-Valued Logic, ISMVL-2008, Dallas, Texas*, pages 106–111, IEEE-CS Press, 2008.

– Josep Argelich, Alba Cabiscol, Inês Lynce and Felip Manyà. Modelling MaxCSP as Partial MaxSAT. In *11th International Conference on Theory and Applica-*

tions of Satisfiability Testing, SAT-2008, Guangzhou, P. R. China, pages 1–14, Springer LNCS 4996, 2008.

– Josep Argelich, Alba Cabiscol, Inês Lynce and Felip Manyà. Sequential Encodings from MaxCSP into Partial MaxSAT. In *12th International Conference on Theory and Applications of Satisfiability Testing, SAT-2009, Swansea, Wales, United Kingdom*, pages 161–166, Springer LNCS 5584, 2009.

– Josep Argelich, Alba Cabiscol, Inês Lynce and Felip Manyà. Regular Encodings from MaxCSP into Partial MaxSAT. In *39th International Symposium on Multiple-Valued Logic, ISMVL-2009, Okinawa, Japan*, pages 196–202, IEEE-CS Press, 2009.

– Ramón Béjar, Alba Cabiscol, Jordi Planes and Felip Manyà. Generating Hard Instances for MaxSAT. In *39th International Symposium on Multiple-Valued Logic, ISMVL-2009, Okinawa, Japan*, pages 191–195, IEEE-CS Press, 2009.

– Josep Argelich, Alba Cabiscol, Inês Lynce and Felip Manyà. New Insights into Encodings from MaxCSP into Partial MaxSAT. In *40th International Symposium on Multiple-Valued Logic, ISMVL-2010, Okinawa, Japan*, pages 46–52, IEEE-CS Press, 2010.

– Josep Argelich, Alba Cabiscol, Inês Lynce and Felip Manyà. Efficient Encodings from CSP into SAT, and from MaxCSP into MaxSAT. *Journal of Multiple-Valued Logic and Soft Computing*, 19: pages 3–23, 2013.

1.4 Outline of the thesis

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

Chapter 2: The SAT Problem. We provide an overview of the most relevant techniques for solving SAT. First, we introduce some basic concepts commonly used

in SAT. Second, we present the resolution method, which applies an inference rule that provides a refutation complete inference system. Third, we describe DP, the first effective method for producing resolution refutations. Fourth, we present the DLL procedure, implemented in the majority of state-of-the-art complete SAT algorithms, and review the main solving techniques that have been incorporated into DLL in order to devise fast SAT solvers. Finally, we describe some representative local search algorithms for SAT.

Chapter 3: The MaxSAT Problem. We introduce MaxSAT formalisms, and review the solving techniques that have proved to be useful in terms of performance. First, we introduce some background definitions in MaxSAT. Second, we present the branch-and-bound schema, which is the one of the most commonly used approach to exact MaxSAT solving, and explain how this schema can be improved with good quality lower bounds. Third, we describe how to solve MaxSAT by solving a sequence of SAT instances, and finally review the most representative SAT-based MaxSAT algorithms.

Chapter 4: Encoding CSP into SAT. We present an overview of the existing encodings from CSP into SAT, and define two new encodings: the minimal support encoding and the interval-based support encoding. First, we introduce some background definitions in CSP. Second, we review the different ways of encoding CSP variables into SAT. Third, we present the direct encoding and the support encoding, which are the most frequently used encodings from CSP into SAT. Then, we present the log encoding. Fourth, we define the minimal support encoding and its variants. Next, we define the interval-based support encoding, which is the first polynomial size support encoding containing only regular literals. Finally, we report on an experimental investigation, and analyze the impact of the encodings on the performance of SAT solvers.

Chapter 5: Encoding MaxCSP into Partial MaxSAT. We extend the encodings from CSP into SAT, described in Chapter 4, for encoding MaxCSP into

MaxSAT. First, we extend the standard encodings from CSP into SAT: the direct, support, minimal support and interval-based support, obtaining its Partial MaxSAT versions. We prove their correctness, as well as some properties of the encodings. Second, we report on an empirical comparison of the defined encodings using both branch-and-bound and SAT-based MaxSAT solvers.

Chapter 6: Generation of Hard MaxSAT Instances. We describe and empirically evaluate MaxSAT instance generators of adjustable hardness for testing MaxSAT solvers. The generators encode into MaxSAT the following combinatorial optimization problems: Max1+pSAT, Partial Max2SAT, MaxCut, and rectangular bin packing. The empirical evaluation of the proposed generators shows that they produce challenging and suitable benchmarks for both branch-and-bound and SAT-based MaxSAT solvers.

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

Chapter 2

The SAT Problem

In this chapter we define the SAT problem and present an overview of solving techniques frequently used in SAT. In Section 2.1, we define the syntax and semantics of CNF formulas. In Section 2.2, we review the most relevant methods for solving SAT. In Section 2.2.1, we describe the most popular complete algorithms: the resolution method [Rob65], the Davis-Putnam procedure (DP) [DP60], and the Davis-Logemann-Loveland procedure (DLL) [DLL62]; and in Section 2.2.2, we describe two well-known local search algorithms: GSAT [SLM92] and WalkSAT [SKC94].

In some parts of this chapter we follow closely the presentation of [Arg08]. The aim of including this chapter is to have a self-contained document, but not provide a scientific contribution.

2.1 Satisfiability preliminaries

Given a finite set of *Boolean variables* $\{x_1, \ldots, x_n\}$, a variable x_i may take values 0 (for false) or 1 (for true). A *literal* ℓ_i is a propositional variable x_i or its negation $\neg x_i$. The *complementary* of a literal ℓ, denoted by $\bar{\ell}$, is x if $\ell = \neg x$ and is $\neg x$ if $\ell = x$.

A *clause* is a disjunction of literals, and a CNF formula is a conjunction of clauses. A *CNF formula* is often represented as a set of clauses.

The *size* of a clause C , denoted by $|C|$, is the total number of literal occurrences in the clause. A clause with one literal is called *unit clause*, with two literals is called

binary clause, and with three literals is called *ternary clause*. The *size* of a CNF formula ϕ, denoted by $|\phi|$, is the sum of the sizes of all its clauses.

A *truth assignment* I is a mapping that assigns to each propositional variable either the value 0 or the value 1. A truth assignment satisfies a literal x_i if x_i takes the value 1, and satisfies a literal $\neg x_i$ if x_i takes the value 0; satisfies a clause if it satisfies at least one literal of the clause; and satisfies a CNF formula if it satisfies all the clauses of the formula. A CNF formula is *satisfiable* if there exists an assignment that satisfies the formula; otherwise, it is *unsatisfiable*. When an assignment does not satisfy a literal (clause, CNF formula), we say that it is falsified.

A CNF formula is a *tautology* if it is satisfied by any truth assignmet. An *empty clause*, denoted by $\square$, is a clause with no literals. An *empty clause* is unsatisfied by any truth assignment. Two CNF formulas are *equivalent* if they are satisfied by the same set of assignments.

A truth assignment is *complete* if all the variables ocurring in the CNF formula ϕ have been assigned; otherwise, it is *partial*.

In a partial truth assignment for a CNF formula, there exits three kind of clauses: the *satisfied* clauses, these clauses contain at least one satisfied literal; the *unsatisfied* clauses, in which all the literals in the clause are falsified, and the *unresolved* clauses, the clauses that the partial assignment makes them not to be decided. The unassigned literals of a clause are referred to as its *free literals*. In a search context, an unresolved clause is said to be *unit* if the number of its free literals is one. Similarly, an unresolved clause with two free literals is said to be *binary*, and an unresolved clause with three free literals is said to be *ternary*.

The Satisfiability Problem (SAT) for a CNF formula ϕ is the problem of deciding if there is a *truth assignment* that satisfies ϕ.

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

$$\begin{aligned} c_1 &: x_1 \vee \neg x_2 \\ c_2 &: x_1 \vee x_3 \\ c_3 &: \neg x_1 \vee x_2 \vee x_3 \end{aligned}$$

Suppose that the we have the following partial truth assignment $\{I(x_1) = 0, I(x_2) = 0\}$. *This means that clauses* c_1 *and* c_3 *are satisfied and clause* c_2 *is unresolved. Notice that clause* c_2 *is also unit because* x_3 *is the only free literal. Therefore, the CNF formula is unresolved.*

Suppose now that this assignment is completed by adding $I(x_3) = 0$. *Then, clause* c_2 *becomes unsatisfied. Finally, if we have the assignment* $\{I(x_1) = 1, I(x_2) = 0, I(x_3) = 1\}$, *all the clauses are satisfied.*

2.2 Satisfiability algorithms

We review the solving techniques which are frequently used in SAT. First, we describe some well-known complete algorithms, and then some local search algorithms. Complete algorithms perform a search through the space of all possible truth assignments, in a systematic manner, to prove either a given formula is satisfiable (the algorithm finds a satisfying truth assignment) or unsatisfiable (the algorithm explores the entire search space without finding any satisfying truth assignment). By contrast, local search algorithms usually do not explore the entire search space, and a given truth asssignment can be considered more than once.

Concerning complete algorithms, Section 2.2.1 starts by presenting the resolution method, which applies an inference rule that provides a refutation complete inference system. Then, it describes DP, the first effective method for producing resolution refutations. Finally, it presents the DLL procedure, implemented in the majority of the state-of-the-art complete SAT algorithms, and reviews the main solving techniques that have been incorporated into DLL in order to devise fast SAT solvers. Concerning local search algorithms, Section 2.2.2 describes basic solving techniques of local search algorithms for SAT, including a description of GSAT and WalkSAT.

2.2.1 Complete algorithms

Resolution Method

Resolution is one of the complete methods used to solve SAT. It is based on the resolution rule, which provides a refutation complete inference system [Rob65].

Given two clauses c_1, c_2, called *parent clauses*, then r is a *resolvent* of c_1 and c_2 if there is one literal $\ell \in c_1$ such that $\bar{\ell} \in c_2$, and r has the form

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

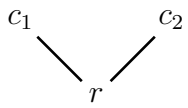

Figure 2.1: Resolution rule: r is a *resolvent* of the *parent clauses* c_1 and c_2

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

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

The resolution procedure consists in computing resolution steps to a formula ϕ until the empty clause is derived or no more new resolvents exist. Then, the formula is unsatisfiable if $\square \in \phi$; otherwise, ϕ is satisfiable. Algorithm 2.1 describes this procedure [Sch89].

The Davis-Putnam procedure

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

1. `Unit Propagation` (UP), also referred to as *Boolean constraint propagation* [ZM88], is the iterated application of the *Unit Clause* (UC) *rule* (also

Algorithm 2.1: `Resolution`(ϕ): Resolution based SAT algorithm

Output: Satisfiability of ϕ

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

```
repeat
    φ' ← φ
    φ ← Res(φ)
until □ ∈ φ ∨ φ = φ'
if □ ∈ φ then return false
else return true
```

referred to as the *one-literal rule*) until an empty clause is derived or there are no unit clauses left. If a clause is unit, then its literal must be assigned the value true. If $\{\ell\}$ is a unit clause of a CNF formula ϕ, UC consists in deleting all the clauses of ϕ with literal ℓ and removing all the occurrences of literal $\bar{\ell}$.

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

3. `Resolution` is applied in order to iteratively eliminate each variable from the CNF formula. In order to do so, DP applies a refinement (a restriction) of the resolution method, known as *variable elimination*: Let $\mathcal{C}_\ell$ be the set of clauses containing ℓ and $\mathcal{C}_{\bar{\ell}}$ the set of clauses containing $\bar{\ell}$, the method consists in generating all the non-tautological resolvents using all clauses in $\mathcal{C}_\ell$ and all clauses in $\mathcal{C}_{\bar{\ell}}$, and then removing all clauses in $\mathcal{C}_\ell \cup \mathcal{C}_{\bar{\ell}}$. After this step, the CNF formula contains neither ℓ nor $\bar{\ell}$.

The pseudo-code of DP is given in Algorithm 2.2. The algorithm selects a variable to be eliminated among the shortest clauses. The worst-case memory requirement for DP is exponential. In practice, DP can only handle SAT instances with tens of variables because of this exponential blow-up [Urq87, CS00]. The procedure stops

Algorithm 2.2: `DavisPutnam`(ϕ) : Davis-Putnam procedure for SAT

Output: Satisfiability of ϕ

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

```
UnitPropagation(φ)
PureLiteralRule(φ)
if φ = ∅ then return true
if □ ∈ φ then return false
ℓ ← literal in c ∈ φ having c the minimum length
R_ℓ ← all possible non-tautological resolvent clauses between all clauses in
C_ℓ and all clauses in C_ℓ̄
return DavisPutnam(φ ∪ R_ℓ \ (C_ℓ ∪ C_ℓ̄))
```

applying resolution when the CNF formula is found to be either satisfiable or unsatisfiable. It is declared to be unsatisfiable whenever a conflict is reached, detected while applying rule UC. If no conflict is reached, the CNF formula becomes empty and is declared to be satisfiable.

Example 2.2 *Given the following CNF formula, we demonstrate its satisfiability using algorithm DP:*

$$x_1 \wedge (x_1 \vee x_2) \wedge (x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_3 \vee x_5) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_5)$$

We show the steps applied by algorithm DP using a table. In the first column, the input formula is displayed, where each line represents a different clause. The rest of the columns represent the result of applying UC. The table below shows the application of the rule to literal x_1. Removed clauses are marked with a '$\times$' and modified clauses

are displayed in bold.

ϕ	x_1
(x_1)	$\times$
$(x_1 \vee x_2)$	$\times$
$(x_2 \vee x_4)$	$(x_2 \vee x_4)$
$(\neg x_1 \vee x_3 \vee \neg x_4)$	$(\mathbf{x_3} \vee \neg \mathbf{x_4})$
$(x_3 \vee x_5)$	$(x_3 \vee x_5)$
$(\neg x_1 \vee \neg x_3 \vee \neg x_5)$	$(\neg \mathbf{x_3} \vee \neg \mathbf{x_5})$

In a second step, DP applies the pure literal rule. The table below shows the application of the rule to literal x_2, and then to literal $\neg x_4$.

ϕ'	x_2	$\neg x_4$
$(x_2 \vee x_4)$	$\times$	
$(x_3 \vee \neg x_4)$	$(x_3 \vee \neg x_4)$	$\times$
$(x_3 \vee x_5)$	$(x_3 \vee x_5)$	$(x_3 \vee x_5)$
$(\neg x_3 \vee \neg x_5)$	$(\neg x_3 \vee \neg x_5)$	$(\neg x_3 \vee \neg x_5)$

Finally, DP applies resolution. The table below shows the elimination of variable x_3. Observe that a tautological clause appears, and is removed by the method.

ϕ''	x_3	
$(x_3 \vee x_5)$	$\times$	
$(\neg x_3 \vee \neg x_5)$	$(x_5 \vee \neg x_5)$	$\times$

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

The Davis-Logemann-Loveland procedure

The vast majority of state-of-the-art complete SAT algorithms are built upon the backtrack search algorithm of Davis, Logemann and Loveland (DLL) [DLL62]. DLL replaces the application of resolution in DP by the splitting of the CNF formula into two subproblems. Given a literal ℓ occurring in ϕ, the first subproblem ($\phi_{\bar{\ell}}$) is the

application of UC over ϕ with $\bar{\ell}$, and the second subproblem (ϕ_ℓ) is the application of UC over ϕ with ℓ. Then, ϕ is unsatisfiable if and only if ϕ_ℓ and $\phi_{\bar{\ell}}$ are unsatisfiable. This method is shown in Algorithm 2.3.

Procedure DLL essentially constructs a binary search tree in a depth-first manner. The leaf nodes not containing empty clauses represent complete assignments (i.e., all variables are assigned) while internal nodes represent partial assignments (i.e., some variables are assigned, the rest are free). The DLL procedure explores the search tree and determines that there exists an assignment that satisfies the input formula if the empty formula is derived, and that there exists no assignment that satisfies the input formula if all the branches of the search tree contain the empty clause.

DLL incorporates `Unit Propagation` and `Pure Literal Rule` in order to avoid the explicit exponential enumeration of the whole search space. Using a variable selection heuristic, the branching variables are selected to reach a dead-end as early as possible.

Example 2.3 *The search tree for the CNF formula below is displayed in Figure 2.2.*

$$\Gamma_0 = \phi \quad : \quad (x_1 \vee x_5) \wedge (x_1 \vee \neg x_6) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee x_6)$$

Solid lines are for splitting assignments, and dashed lines are for unit propagation and monotone literal assignments. Black nodes mark reached conflicts.

The subproblem associated with each internal node are as follows:

Algorithm 2.3: `DavisLogemannLoveland`(ϕ) : DLL procedure for SAT

Output: Satisfiability of ϕ

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

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

$$
\begin{aligned}
\Gamma_{10} = \Gamma_{0(\neg x_1)} &: (x_5), (\neg x_6), (\neg x_2 \vee x_4), (x_2 \vee \neg x_4), (\neg x_2 \vee \neg x_4 \vee \neg x_5), (x_2 \vee x_4 \vee \neg x_3), \\
&\quad (x_2 \vee x_3 \vee x_6) \\
\Gamma_{11} = \Gamma_{0(x_1)} &: (\neg x_2 \vee \neg x_4 \vee \neg x_5), (x_2 \vee x_4 \vee \neg x_3), (\neg x_2), (x_2 \vee x_3 \vee x_6) \\
\Gamma_{20} = \Gamma_{10(x_5)} &: (\neg x_6), (\neg x_2 \vee x_4), (x_2 \vee \neg x_4), (\neg x_2 \vee \neg x_4), (x_2 \vee x_4 \vee \neg x_3), (x_2 \vee x_3 \vee x_6) \\
\Gamma_{21} = \Gamma_{11(\neg x_2)} &: (x_4 \vee \neg x_3), (x_3 \vee x_6) \\
\Gamma_{30} = \Gamma_{20(\neg x_6)} &: (\neg x_2 \vee x_4), (x_2 \vee \neg x_4), (\neg x_2 \vee \neg x_4), (x_2 \vee x_4 \vee \neg x_3), (x_2 \vee x_3) \\
\Gamma_{31} = \Gamma_{21(x_4)} &: (x_3 \vee x_6) \\
\Gamma_{40} = \Gamma_{30(\neg x_2)} &: (\neg x_4), (x_4 \vee \neg x_3), (x_3) \\
\Gamma_{41} = \Gamma_{30(x_2)} &: (x_4), (\neg x_4) \\
\Gamma_{50} = \Gamma_{40(x_3)} &: (\neg x_4), (x_4)
\end{aligned}
$$

Figure 2.2 shows that the input CNF formula is satisfiable because all the variables have been assigned and the empty formula is reached.

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

1. DP increases the number and length of the clauses rather quickly. DLL never increases the length of clauses.

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

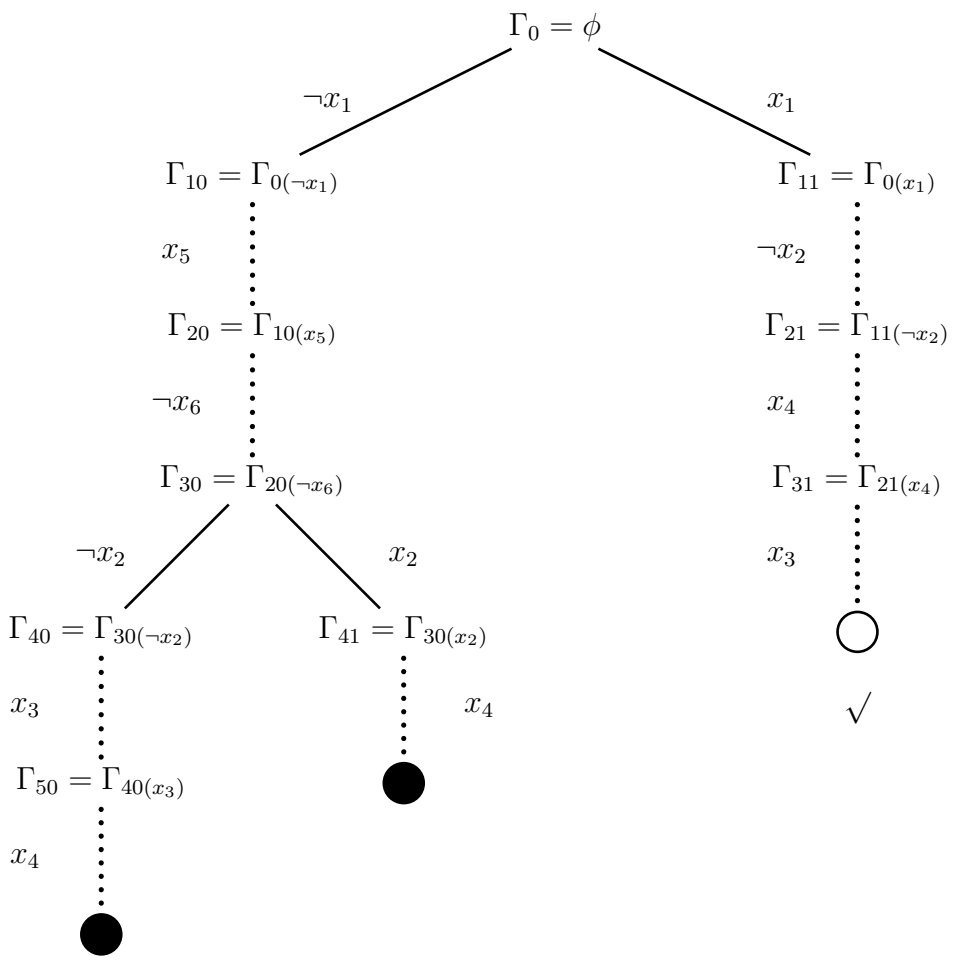

Figure 2.2: Search tree for DLL applied to Example 2.3.

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

Solving techniques for improving DLL

In this section we focus on important solving techniques that should be taken into account when developing SAT solvers that implement the DLL procedure: the variable selection heuristic, the data structures, clause learning, application of restarts, and

reasoning on special structures. Our description of clause learning follows closely the presentation of [Zha03].

Among the most relevant complete algorithms developed in the last years based on the DLL procedure, we highlight the following ones:

CryptoMiniSat [Soo09] by Mate Soos.

Glucose [AS09] by Gilles Audemard and Laurent Simon.

GRASP [MSS99] by João Marques-Silva and Karem Sakallah.

MiniSat [ES03] by Niklas Eén and Niklas Sörensson.

Plingeling [Bie10a] by Armin Biere.

PrecoSAT [Bie10b] by Armin Biere.

Satz [LA97b] by Chu Min Li and Anbulagan.

zChaff [MMZ+01] by Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang and Sharad Malik.

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

The original variable selection heuristic in DLL selects a variable occurring in clauses of minimum size. The variable is selected after applying unit propagation and the pure literal rule, and is used to split the CNF formula into two subproblems.

Example 2.4 *Let ϕ be the following CNF formula:*

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_4 \vee \neg x_3) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee x_4 \vee x_6) \wedge (\neg x_2 \vee x_4 \vee x_6) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

The shortest clauses in ϕ are $(\neg x_1 \vee x_2)$ and $(x_1 \vee \neg x_5)$, hence the heuristic of DLL chooses any of the following variables: x_1, x_2 or x_5.

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

Example 2.5 *Let ϕ be the CNF formula of Example 2.4. The shortest clauses in ϕ are $(\neg x_1 \vee x_2)$ and $(x_1 \vee \neg x_5)$, hence MOMS heuristic chooses variable x_1 because it occurs twice.*

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

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

where $|c|$ is the number of literals in clause c. JW chooses a variable x that maximizes $J(x) + J(\neg x)$.

Example 2.6 *Let ϕ be the CNF formula of Example 2.4. With function J, one can get: $J(x_1) = 0.25$, $J(\neg x_1) = 0.25$,incomplete $J(x_2) = 0.5$, $J(\neg x_2) = 0.25$, $J(x_3) = 0$, $J(\neg x_3) = 0.25$, $J(x_4) = 0.5$, $J(\neg x_4) = 0$, $J(x_5) = 0$, $J(\neg x_5) = 0.25$, $J(x_6) = 0.25$, $J(\neg x_6) = 0$. The variable x_2 with $J(x_2)+J(\neg x_2) = 0.75$ is chosen by heuristic JW.*

Several heuristics based on unit propagation have been proved useful and allow to exploit yet more the power of unit propagation; e.g., the heuristics of POSIT [Fre95], Tableau [CA96] and Satz [LA97a]. A unit propagation heuristic works as follows: Given a variable x, it examines x by respectively adding the unit clause x and $\neg x$ to a CNF formula, and independently computes two unit propagations. The real effect of the unit propagations is then used to weight x and detect failed literals. A *failed literal* is a literal whose addition to a CNF formula brings the empty clause after unit propagation.

Given a CNF formula ϕ, this heuristic requires propagating a literal x to count the clauses reduced in the subproblem obtained, and to follow the same process with the complementary literal $\neg x$. Let $w(x)$ and $w(\neg x)$ be the number of clauses reduced by x and $\neg x$ respectively. This heuristic consist in choosing the variable which maximizes at the same time $w(x)$ and $w(\neg x)$, maximizing the following function: $F(x) = w(x) * w(\neg x) * 1024 + w(x) + w(\neg x)$. If literal x ($\neg x$) is a failed literal, then $\neg x$ (x) is fixed. This approach makes possible to better prevent the consequences that the choice of the literal will produce.

Example 2.7 *Let ϕ be the CNF formula of Example 2.4. The application of function w to each literal produces the following values: $w(x_1) = 3$, $w(\neg x_1) = 1$, $w(x_2) = 2$, $w(\neg x_2) = 4$, $w(x_3) = 2$, $w(\neg x_3) = 0$, $w(x_4) = 0$, $w(\neg x_4) = 4$, $w(x_5) = 4$, $w(\neg x_5) = 0$, $w(x_6) = 0$, $w(\neg x_6) = 2$. The variable x_2 with $w(x_2) = 2, w(\neg x_2) = 4$ is chosen by a unit propagation based heuristic.*

However, since examining a variable by two unit propagations is time consuming, two major problems remain open: should one examine all the free variable by unit propagation at every node of a search tree? Otherwise, what are the variables to be examined at a search tree node? In [LA97a], the authors try to experimentally address these two questions in order to obtain an optimal exploitation. They define a predicate $PROP_z$ at a search tree node whose

meaning is the set of variables that will be examined at the node; i.e., variable x is examined if $PROP_z(x)$ is true.

$PROP_z$ is defined as follows: if there are more than T (parameter empirically set to 10) variables occurring both negatively and positively in binary clauses and having at least 4 binary occurrences, then only all these variables are examined; otherwise, if there are more than T variables occurring both negatively and positively in binary clauses and having at least 3 binary occurrences, then only all these variables are examined; otherwise all the free variables are examined.

Another approach consists in selecting a variable that is likely to be a backbone variable [DD01, KSTW05]. A variable is a backbone variable if the variable has assigned the same value in all the solutions. Given a CNF formula ϕ, this heuristic tries first on variables that belong (or are expected to belong) to the backbone of ϕ. If backbone variables are selected first, the algorithm searches through fewer branches, speeding up the solver. The heuristic of Satz and the heuristics based on the notion of backbone are quite effective on computationally difficult random SAT instances.

The previous heuristics were created without the addition of learning techniques into SAT solvers. The two following heuristics are thought for this kind of solvers, focusing on a kind of locality rather than focusing on formula simplification [Mit05]. For the solver zChaff [MMZ$^+$01, ZM02, Zha03], the authors proposed a branching heuristic called Variable State Independent Decaying Sum (VSIDS). This heuristic keeps a score for each literal. Initially, the score is the number of occurrences of the literal in the initial problem. Because of the learning mechanism, clauses are added to the formula as the search progresses. VSIDS increases the score of a literal by a constant whenever an added clause contains the literal. More than to develop an accurate heuristic, the motivation was to design a fast and dynamically adapting heuristic.

The SAT solvers BerkMin [GN01] and siege [Rya04] have improved the VSIDS heuristic. BerkMin also measures the age of the clauses and the activity for

deciding the next branching variable, whereas siege gives priority to assigning variables on recently recorded clauses.

Efficient data structures The performance of the DLL procedure critically depends upon the care taken in the implementation. Solvers implementing DLL spend much of their time applying unit propagation [Zha97, LMS02], and this has motivated the definition of several proposals to reduce the cost of applying unit propagation.

The simplest and more intuitive implementation of unit propagation is to keep counters for each clause. This schema is attributed to Crawford and Auton [CA93] by [ZS96]. Similar schemas were since then employed in many solvers such as GRASP [MSS99], Relsat [BS97] and Satz [LA97a]. For example, in GRASP each clause keeps two counters, one for the satisfied literals in the clause and another for the unsatisfied literals in the clause. Each variable has two lists that contain all the clauses in which that variable appears with positive and negative polarity. When a variable is assigned a value, the counters of the clauses that contain this variable are updated. If a counter of unsatisfied literals becomes equal to the total number of literals in the clause, then the clause is conflicting. If a counter of unsatisfied literals is one less than the total number of literals in the clause and the counter of satisfied literals is null, then the clause is a unit clause. A counter-based unit propagation procedure is easy to understand and implement, but this schema may be improved.

As it is pointed out in [ZM02], Zhang and Stickel [ZS96], in order to speed up the application of unit propagation, created a new data structure in the solver SATO: head/tail lists. In this data structure, each clause has two pointers associated with it, called the head and the tail pointer. A clause stores all its literals in an array. Initially, the head pointer points to the first literal of the clause and the tail pointer points to the last literal of the clause. Each variable keeps four linked lists that contain pointers to clauses. Each of these lists contains the pointers to the clauses that have their head/tail literal in

positive/negative polarity for a given variable. Whenever a variable is assigned, only two of the four lists are examined. The head/tail list schema is faster than the counter-based schema because when the variable is assigned the value true (false), the clauses that contain the variable with positive (negative) polarity are not visited. For both the counter-based algorithm and the head/tail list-based algorithm, undoing a variable assignment during backtrack has about the same computational complexity as assigning the variable.

The solver zChaff implements a unit propagation algorithm based on the so-called 2-literal watching schema. Similar to the head/tail list schema, 2-literal watching also has two special literals, called watched literals, for each clause. Each variable has two lists containing pointers to all the watched literals containing this variable in either polarity. In contrast to the head/tail list schema of SATO, there is no imposed order on the two pointers within a clause, and each of the pointers can move in either direction. In addition, no references have to be kept to the just assigned literals, since pointers do not move when backtracking is applied. This data structure was also used in the solvers BerkMin [GN01] and MiniSat [ES03].

The main problem of pointer-based data structures is that they cannot keep precise information about clauses with more than two free literals. The inclusion of additional literal references as a solution has been referred in [Gel02], and techniques to rearrange the list of literals have been investigated in [LMS05, Nad02].

Clause learning and non-chronological backtracking When a conflicting clause is found, a SAT solver finds out the reason of the conflict and tries to solve it by applying a *conflict analysis* procedure. This procedure tells the SAT solver that there exists no solution in the search space with the current partial assignment, and indicates a new search space to continue the search for a solution. The assigned variables are categorized as decision variables (i.e., picked using a variable selection heuristic) or propagation variables (i.e.,

assigned using unit propagation). The *decision level* of variable x is the number of decision variables in an assignment that were assigned before x. We call *conflict level* at the decision level at which a conflict occurs.

The simplest conflict analysis procedure is known as *chronological backtracking*, and is applied in the original DLL procedure. When a conflict is detected, the search backtracks to the most recent decision level with a variable that has not been flipped. Chronological backtracking has good performance on random instances and is used in SAT solvers like Satz [LA97a].

Modern SAT solvers apply a more sophisticated conflict analysis procedure, called *non-chronological backtracking* or *conflict direct backjumping*, that can get more information about the conflict. This procedure detects the reason of the conflict and often backtracks to a smaller decision level which produces the conflict. Example 2.8 shows the difference between chronological and non-chronological backtracking.

Example 2.8 *Let us consider a CNF formula ϕ with the following clauses among others:*

$$\begin{array}{lcc} c_1 & : & x_4 \vee x_8 \vee x_9 \\ c_2 & : & x_4 \vee x_8 \vee \neg x_9 \\ c_3 & : & x_4 \vee \neg x_8 \vee x_9 \\ c_4 & : & x_4 \vee \neg x_8 \vee \neg x_9 \\ \vdots & : & \vdots \\ c_m & : & \ldots \end{array}$$

and lexicographical order as variable selection heuristic. Suppose we assign all the variables to $false$ until decision level 7 without finding conflicts. When we reach the decision level 8, we detect a contradiction when the variable x_8 is set to $false$ between clauses c_1 and c_2, and between clauses c_3 and c_4 if we set variable x_8 to $true$. A chronological backtracking solver would backtrack to decision level 7 because it is the previous decision level with a variable that has not been flipped. However, flipping variables at a decision level greater than 4 does not resolve

the conflict. A non-chronological backtracking solver can analyze this particular problem, determine the variable that produces the conflict, and backtrack to its level. In this example, the conflict analysis procedure would resolve to backtrack to level 4 and flip variable x_4 to true.

During the conflict analysis process, the information about the current conflict can be stored by means of redundancy [BS94, BGS99]. These redundant clauses do not change the satisfiability of the original formula, and they help prune parts of the search space with conflicts that involve variables of the *learned* conflict. This technique is called *clause learning* or *conflict driven clause learning*, and is used in solvers like BerkMin [GN01], Chaff [MMZ+01], GRASP [MSS99], MiniSat [ES03] and siege [Rya04].

The implication relationships of variable assignments during the SAT solving process can be represented in an *implication graph*. An implication graph is a directed acyclic graph (DAG) in which each node represents a variable assignment, and the incident edges of a vertex are the reasons that imply the variable assignment. Figure 2.3 shows a typical implication graph. The incident edges to node x_5 are from x_1 and $\neg x_4$, which means that if x_1 is $true$ and x_4 is $false$, then x_5 must be $true$. A decision vertex has no incident edge. In an implication graph, a variable and its negation only appear when a conflict occurs. Such a variable is called *conflicting variable*. The conflicting variable in Figure 2.3 is x_6.

In SAT solvers, clause learning can be applied by analyzing the implication graph. For example, in Figure 2.3, by examining the incident vertex of the nodes of the conflict variable, it is easy to see that the assignment of x_2 and x_4 to $false$, and x_3 and x_5 to $true$ leads to the conflict between nodes x_6 and $\neg x_6$:

$$\neg x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5 \Rightarrow conflict$$

If we want to avoid the conflict we could add the following clause:

$$\neg(\neg x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5) \Leftrightarrow x_2 \vee \neg x_3 \vee x_4 \vee \neg x_5$$

As a result we get the *conflict clause* that is represented by the *cut* labeled as *conflict* in the implication graph.

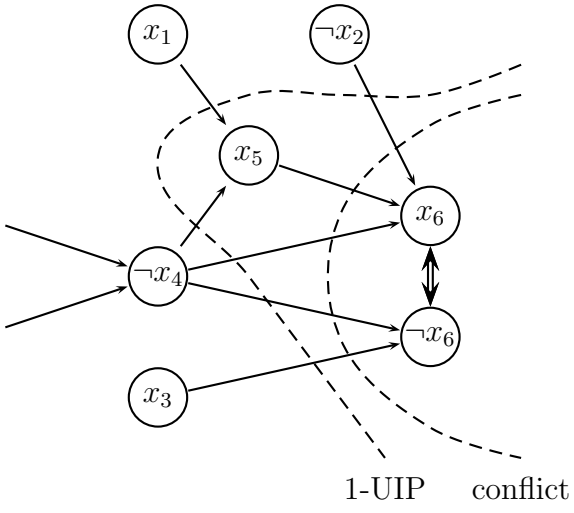

Figure 2.3: Implication graph.

The example shows that we can generate a conflict clause doing a bipartition of the implication graph. In the bipartition, we have decision variables on one side (the reason side), and conflicting variables on the other side (the conflict side). Each variable on the reason side with an edge to the conflict side belongs to the learned clause. This bipartition is called a *cut*, and different cuts correspond to different learning schemas.

Many learning schemas have been studied in the literature and L. Zhang compares some of them in his Ph.D. Thesis [Zha03]. One of the learning schemas with better performance, and used in modern SAT solvers, is 1-UIP (*first Unique Implication Point*) [MSS99]. A UIP is a vertex that dominates both vertices corresponding to the conflicting variable. The UIP is not unique, and we can find more than one in an implication graph. The 1-UIP learning schema picks the UIP closer to the conflict and cuts just after it. In Figure 2.3, there is only one UIP, represented by vertex $\neg x_4$, and the 1-UIP learning schema is represented by the cut labeled 1-UIP. The learned clause of this cut is $\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4$.

The number of learned clauses can increase drastically the size of the database of clauses. Several clause deletion strategies have been proposed [GN01, BS97] but it is still an open topic.

Reasoning on special structures in SAT problems. Given that many problems like pigeonhole or graph coloring involve a great deal of symmetry in their arguments, there has been suggested to add so-called symmetry-breaking clauses to the original formula [CGLR96, ASM06]; these are clauses that break the existing symmetry without affecting the overall satisfiability of the formula. Rather than modifying the set of clauses in the problem, it is also possible to modify the notion of inference, so that once a particular conflict has been derived, symmetric equivalents can be derived in a single step [Kri85].

Another explored deduction mechanism for special structured instances is equivalence reasoning. Solver eqsatz [Li03] incorporated equivalence reasoning into the Satz solver and found that it is effective on some particular classes of benchmarks. In that work, the equivalence reasoning is accomplished by a pattern-matching schema for equivalence clauses. In particular, finding equivalences of the type $x_1 \leftrightarrow x_2$ can reduce the number of variables and clauses of the formula, since variables x_1 and x_2 can be collapsed into one variable. A related deduction mechanism was proposed in [LMS01]. There, the authors propose to include more patterns in the matching process for simplification purposes in deduction. A more complex equivalence reasoning, with several steps, is performed in [WvM98, HDvMvZ04] as a preprocessing.

2.2.2 Incomplete algorithms

Local search for SAT

Complete algorithms explore, in a systematic manner, the entire search space in order to prove the satisfiability of a given formula. So, one of the problems of complete methods (e.g., DP and DLL),is their inability to solve hard random 3-SAT instances with

more than 700 propositional variables within a reasonable amount of time [DD01].This problem can be overcomed using incomplete methods. An incomplete method in SAT can find a satisfying assignment, but cannot prove the unsatisfiability of a CNF formula. If a solution is found, the formula is declared satisfiable and the algorithm terminates successfully; but if the algorithm fails to find a solution, no conclusion can be drawn. The most known local search methods in SAT are GSAT [SLM92] and WalkSAT [SKC94]. By contrast, these procedures are able to solve hard instances with more than 100,000 variables, though completeness is lost.

Local search methods start typically with some ramdomly generated complete assignment and try to find a satisfying assignment by iteratively changing the assignment of one propositional variable. Each change of the assignment of a variable is called a variable flip, and variables are selected heuristically. Such changes are repeated until either a satisfying assignment is found or a pre-set maximun number of changes is reached.This process is repeated as needed, up to a pre-set number of times. This allows to explore the search space moving from one search space position to a neighboring position. The decision on each step (change) is based on information about local neighborhood only. Usually, local search algorithms do not explore the entire search space, and a given assignment may be considered more than once.

The main difference among the different local search algorithms for SAT lies in the strategy used to select the variable to be flipped next. Furthermore, local search algorithms can get trapped in local minima and plateau regions of the search space, leading to premature stagnation of the search. One of the simplest mechanisms for avoiding premature stagnation of the search is random restart, which reinitializes the search if no solution has been found after a fixed number of steps. Random restarts are used in almost every local search algorithm for SAT.

A general outline of a local search algorithm for SAT is given in Algorithm 2.4. The generic procedure initializes the search at some complete truth assignment, and then iteratively selects a variable according to the input CNF formula and the current assignment, and flips the selected variable. If after a maximum of $maxSteps$ flips

Algorithm 2.4: `LocalSearch`(ϕ) : General local search procedure

Output: Satisfying assignment of ϕ or 'no solution found'

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

no solution is found, the algorithm restarts from a new randomly generated initial assignment. If after a given number $maxTries$ of such tries still no solution is found, the algorithm terminates unsuccessfully.

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

GSAT algorithm

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

The basic GSAT algorithm uses the following instantiation of the procedure *chooseVariable*(ϕ, A): In each local search step, one of the variables with maximal score is flipped. If there are several variables with maximal score, one of them is randomly selected according to a uniform distribution.

WalkSAT algorithm

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

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

Other local search algorithms

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

HSAT [GW93] by Ian Gent and Toby Walsh.

TSAT [MSG97] by Bertrand Mazure, Lakhdar Saïs and Eric Grégoire.

novelty [MSK97] by McAllester, Selman, Kautz.

novelty+ [Hoo99] by Holger Hoos.

SAPS [HTH02] by Holger Hoos et al.

g2wsat [LH05] by Chu Min Li et al.

Most of these techniques can be checked in solver UBCSAT [TH05], from the University of British Columbia (UBC).

Other local search algorithms developed recently are the following ones:

adaptg2wsat2011 [LWZ07] by Chumin Li, Yu Li and Wanxia Wei.

EBGlucose [Mat11a] by Bryan Matsuo.

EBMiniSat [Mat11b] by Bryan Matsuo.

2.3 Summary

We have presented an overview of the solving techniques most commonly used in SAT. Firstly, basic concepts in satisfiability testing have been introduced. Secondly, complete algorithms for SAT solving such as the DP procedure and the DLL procedure, as well as efficient techniques implemented in complete SAT solvers, are presented. Finally, two of the most representative local search algorithms have been described.

Chapter 3

The MaxSAT Problem

In this chapter we introduce the MaxSAT problem and present an overview of the solving techniques implemented in modern exact MaxSAT solvers. In Section 3.1, we define basic concepts, and the extensions of MaxSAT known as Weighted MaxSAT, Partial MaxSAT and Weighted Partial MaxSAT. In Section 3.2, we give a brief description of the MaxSAT solvers that participated in the 2010 MaxSAT Evaluation. In Section 3.3, we describe how the branch-and-bound schema is adapted to solve MaxSAT, and explain in detail the methods most frequently used to compute lower bounds. In Section 3.4, we describe how to solve MaxSAT by solving a sequence of SAT instances, and review the most representative SAT-based MaxSAT solvers. In some parts of this chapter we follow closely the presentation of [LM09].

3.1 MaxSAT preliminaries

Given a finite set of *Boolean variables* $\{x_1, \ldots, x_n\}$, a variable x_i may take the value 0 (for false) or the value 1 (for true). A literal l_i is a variable x_i or its negation $\neg x_i$. A clause is a disjunction of literals, and a CNF formula is a collection of clauses.

In SAT, a CNF formula is considered to be a set of clauses while, in MaxSAT, a CNF formula is considered to be a multiset of clauses, because repeated clauses cannot be collapsed into a unique clause. For instance, the multiset $\{x_1, \neg x_1, \neg x_1, x_1 \vee x_2, \neg x_2\}$, where a clause is repeated, has a minimum of two unsatisfied clauses. If we

consider the set $\{x_1, \neg x_1, x_1 \vee x_2, \neg x_2\}$, where repeated clauses are collapsed, then it has a minimum of one unsatisfied clause.

A *weighted clause* is a pair (C, w), where C is a clause and w is its weight. The weight can be a natural number or infinity. Its meaning is the penalty (cost) for falsifying the clause C. A clause is *hard* if its corresponding weight is infinity, otherwise the clause is *soft*. A *weighted CNF formula* ϕ is a multiset of weighted clauses, $\phi = \{(C_1, w_1), \ldots, (C_m, w_m)\}$. The *length* of a (weighted) clause is the total number of literal occurrences in the clause. A (weighted) clause with one literal is called *unit*, with two literals is called *binary*, and with three literals is called *ternary*. The *size* of a (weighted) CNF formula ϕ, denoted by $|\phi|$, is the sum of the lengths of all its clauses.

A *weighted partial CNF formula* ϕ is a multiset of weighted clauses $\phi = \{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+k}, \infty)\}$, where the first m clauses are soft, and the last k clauses are hard. If the weight of the soft clauses of ϕ is 1, then we say that ϕ is a Partial CNF formula.

A *truth assignment* I is a mapping that assigns to each propositional variable either the value 0 or the value 1. A truth assignment I satisfies a literal x_i if x_i takes the value 1 and satisfies a literal $\neg x_i$ if x_i takes the value 0, satisfies a clause if it satisfies at least one literal of the clause, and satisfies a CNF formula if it satisfies all the clauses of the formula. A CNF formula is *satisfiable* if there exists an assignment that satisfies the formula; otherwise, it is *unsatisfiable*. An *empty clause*, denoted by $\square$, contains no literals and cannot be satisfied. A *tautology* is a CNF formula that is satisfied by any truth assignment.

A truth assignment I satisfies a weighted clause (C_i, w_i) if it satisfies C_i, and satisfies a weighted CNF formula $\{(C_1, w_1), \ldots, (C_m, w_m)\}$ if it satisfies $C_1, \ldots, C_m$.

Given a CNF formula ϕ, the SAT problem consists in deciding whether there exist a satisfying assignment for ϕ, and the MaxSAT problem consists in finding an assignment that maximizes the number of satisfied clauses in ϕ, or equivalently, that minimizes the number of falsified clauses.

Example 3.1 *Let us consider a MaxSAT instance ϕ with the following clauses:*

$$\begin{aligned} c_1 &: x_1 \vee x_2 \\ c_2 &: \neg x_1 \\ c_3 &: \neg x_1 \vee \neg x_2 \\ c_4 &: \neg x_2 \vee x_3 \\ c_5 &: x_1 \vee \neg x_2 \\ c_6 &: \neg x_3 \end{aligned}$$

An optimal solution for ϕ is the assignment $\{I(x_1) = 1, I(x_2) = 0, I(x_3) = 0\}$, which satisfies 5 clauses. The clause falsified by this assignment is c_2.

We will consider several extensions of the MaxSAT problem which are more well-suited for representing and solving over-constrained problems: Weighted MaxSAT, Partial-MaxSAT, and Weighted Partial MaxSAT.

The *Weighted MaxSAT* problem for a weighted CNF formula ϕ is the problem of finding an assignment that minimizes the sum of weights associated with unsatisfied clauses (or equivalently, that maximizes the sum of weights associated with satisfied clauses) in ϕ.

Example 3.2 *Let us consider a Weighted MaxSAT instance ϕ having the following clauses:*

$$\begin{aligned} c_1 &: (x_1 \vee x_2; 4) \\ c_2 &: (\neg x_1; 3) \\ c_3 &: (\neg x_1 \vee \neg x_2; 5) \\ c_4 &: (\neg x_2 \vee x_3; 3) \\ c_5 &: (x_1 \vee \neg x_2; 2) \\ c_6 &: (\neg x_3; 1) \end{aligned}$$

An optimal solution for ϕ is the assignment $\{I(x_1) = 0, I(x_2) = 1, I(x_3) = 1\}$, which maximizes the sum of weights of satisfied clauses or, equivalently, that minimizes the sum of weights of falsified clauses. The maximum sum of weights of satisfied clauses is 15, and the minimum sum of weights of falsified clauses is 3. This assignment falsifies the clauses c_5 and c_6.

The *Partial MaxSAT problem* for a Partial CNF formula ϕ is the problem of finding an assignment that satisfies all the hard clauses and the maximum number of soft clauses in ϕ. Hard clauses are represented between square brackets, and soft clauses are represented between round brackets.

Example 3.3 *Let us consider a Partial MaxSAT instance ϕ having the following clauses:*

$$\begin{aligned} c_1 &: [x_1 \vee x_2] \\ c_2 &: [\neg x_1] \\ c_3 &: [\neg x_1 \vee \neg x_2] \\ c_4 &: (\neg x_2 \vee x_3) \\ c_5 &: (x_1 \vee \neg x_2) \\ c_6 &: (\neg x_3) \\ c_7 &: (x_1 \vee \neg x_2 \vee \neg x_3) \end{aligned}$$

An optimal solution for ϕ is the assignment $\{I(x_1) = 0, I(x_2) = 1, I(x_3) = 0\}$, which satisfies all the hard clauses and maximizes (minimizes) the number of satisfied (falsified) soft clauses. The maximum (minimum) number of satisfied (falsified) soft clauses is 2 (2), and the satisfied (falsified) soft clauses are c_6 and c_7 (c_4 and c_5).

The *Weighted Partial MaxSAT* problem is the combination of Weighted MaxSAT and Partial MaxSAT. The *Weighted Partial MaxSAT* problem for a *weighted partial CNF formula* ϕ is the problem of finding an assigment that satisfies all the hard clauses and minimizes the sum of weights associated with unsatisfied soft clauses in ϕ (or equivalently, that maximizes the sum of weights associated to satisfied soft clauses).

Example 3.4 *Let us consider a Weighted Partial MaxSAT instance ϕ having the*

following clauses:

$$
\begin{aligned}
c_1 &: [x_1 \vee x_2] \\
c_2 &: [\neg x_1] \\
c_3 &: [\neg x_1 \vee \neg x_2] \\
c_4 &: (\neg x_2 \vee x_3; 3) \\
c_5 &: (x_1 \vee \neg x_2; 2) \\
c_6 &: (\neg x_3; 1) \\
c_7 &: (x_1 \vee \neg x_2 \vee \neg x_3; 5)
\end{aligned}
$$

An optimal solution for the Weighted Partial MaxSAT instance is the assignment $\{I(x_1) = 0, I(x_2) = 1, I(x_3) = 0\}$*, which satisfies all the hard clauses and maximizes (minimizes) the sum of weights of satisfied (falsified) soft clauses. The maximum (minimum) sum of weights of satisfied (falsified) soft clauses is 6 (5), and the satisfied (falsified) soft clauses are* c_6 *and* c_7 *(*c_4 *and* c_5*).*

The MaxSAT problem can also be defined as Weighted MaxSAT restricted to formulas whose clauses have weight 1, and as Partial MaxSAT in the case that all the clauses are declared to be soft. The Partial MaxSAT problem is Weighted Partial MaxSAT when the weights of soft clauses are equal. Notice that the SAT problem is equivalent to Partial MaxSAT when there are no soft clauses.

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

Finally, we introduce the integer linear programming (ILP) formulation of Weighted MaxSAT. Let $\phi = (C_1, w_1) \wedge \cdots \wedge (C_m, w_m)$ be a Weighted MaxSAT instance over the propositional variables $x_1, \ldots, x_n$. With each propositional variable x_i, we associate a variable $y_i \in \{0, 1\}$ such that $y_i = 1$ if variable x_i is true and $y_i = 0$, otherwise. With each clause C_j, we associate a variable $z_j \in \{0, 1\}$ such that $z_j = 1$ if clause C_j is satisfied and $z_j = 0$, otherwise. Let I_j^+ be the set of indices of unnegated

variables in clause C_j, and let I_j^- be the set of indices of negated variables in clause C_j. The ILP formulation of the Weighted MaxSAT instance ϕ is defined as follows:

$$\max F(y,z) = \sum_{j=1}^{m} w_j z_j$$

subject to

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq z_j \quad j = 1, \ldots, m$$

$$y_i \in \{0,1\} \quad i = 1, \ldots, n$$

$$z_j \in \{0,1\} \quad j = 1, \ldots, m$$

If we consider the minimization version of weighted MaxSAT, we assume that, with each clause C_j, we associate a variable $z_j \in \{0,1\}$ such that $z_j = 1$ if clause C_j is falsified and $z_j = 0$, otherwise. Then, the ILP formulation of the instance ϕ is defined as follows:

$$\min F(y,z) = \sum_{j=1}^{m} w_j z_j$$

subject to

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) + z_j \geq 1 \quad j = 1, \ldots, m$$

$$y_i \in \{0,1\} \quad i = 1, \ldots, n$$

$$z_j \in \{0,1\} \quad j = 1, \ldots, m$$

Example 3.5 *The minimization version of the ILP formulation for the Weighted MaxSAT instance ϕ of Example 3.2 is as follows:*

$$\min F(y,z) = 5z_1 + 4z_2 + 3z_3 + 2z_4 + 4z_5 + 1z_6 + 2z_7$$

subject to

$$
\begin{aligned}
& y_1 + z_1 \geq 1 \\
& (1 - y_1) + y_2 + z_2 \geq 1 \\
& y_1 + (1 - y_2) + y_3 + z_3 \geq 1 \\
& (1 - y_1) + (1 - y_2) + z_4 \geq 1 \\
& y_1 + y_2 + (1 - y_3) + z_5 \geq 1 \\
& (1 - y_1) + y_3 + z_6 \geq 1 \\
& (1 - y_1) + (1 - y_2) + (1 - y_3) + z_7 \geq 1 \\
& y_i \in \{0, 1\} \quad i = 1, 2, 3 \\
& z_j \in \{0, 1\} \quad j = 1, 2, 3, 4, 5, 6, 7
\end{aligned}
$$

An assignment that minimizes $F(y, z)$ is: $y_1 = 1$, $y_2 = 1$ and $y_3 = 0$. The variables z_i that must be assigned to 1 are z_4 and z_6.

The ILP formulation of the Weighted Partial MaxSAT instance

$$\phi = [C_1] \wedge \cdots \wedge [C_k] \wedge (C_{k+1}, w_{k+1}) \wedge \cdots \wedge (C_m, w_m)$$

is defined as follows:

$$\max F(y, z) = \sum_{j=k+1}^{m} w_j z_j$$

subject to

$$
\begin{aligned}
& \sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq 1 \quad j = 1, \ldots, k \\
& \sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq z_j \quad j = k+1, \ldots, m \\
& y_i \in \{0, 1\} \quad i = 1, \ldots, n \\
& z_j \in \{0, 1\} \quad j = k+1, \ldots, m
\end{aligned}
$$

If we consider the minimization version of Weighted Partial MaxSAT, then the ILP formulation of the instance ϕ is defined as follows:

$$\min F(y, z) = \sum_{j=k+1}^{m} w_j z_j$$

subject to

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq 1 \quad j = 1, \ldots, k$$

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) + z_j \geq 1 \quad j = k+1, \ldots, m$$

$$y_i \in \{0, 1\} \quad i = 1, \ldots, n$$

$$z_j \in \{0, 1\} \quad j = k+1, \ldots, m$$

Example 3.6 *The minimization version of the ILP formulation for the Weighted Partial MaxSAT instance ϕ of Example 3.4 is as follows:*

$$\min F(y, z) = 1z_1 + 2z_2 + 5z_3 + 6z_4 + 3z_5$$

subject to

$$\begin{aligned}
&y_1 + y_2 \geq 1 \\
&(1 - y_1) + (1 - y_2) \geq 1 \\
&y_1 + (1 - y_2) + y_3 + z_1 \geq 1 \\
&(1 - y_1) + y_2 + y_3 + z_2 \geq 1 \\
&y_2 + (1 - y_3) + z_3 \geq 1 \\
&(1 - y_2) + (1 - y_3) + z_4 \geq 1 \\
&y_3 + z_5 \geq 1
\end{aligned}$$

$$y_i \in \{0, 1\} \quad i = 1, 2, 3$$

$$z_j \in \{0, 1\} \quad j = 1, 2, 3, 4, 5$$

An assignment that minimizes $F(y, z)$ is: $y_1 = 0$, $y_2 = 1$ and $y_3 = 0$. The variables z_i that must be assigned to 1 are z_1 and z_5.

The linear programming (LP) relaxation of the previous formulations is obtained by allowing the integer variables to take real values in $[0, 1]$.

3.2 MaxSAT algorithms

A new generation of exact MaxSAT solvers has been developed in recent years, in part due to the organitzation of an international evaluation of MaxSAT solvers since 2006 [ALMP08, ALMP11a, ALMP11c]. In the 2010 MaxSAT Evaluation, a total of 17 exact MaxSAT solvers were submitted by 18 researchers from different research groups. Table 3.1 contains the list of solvers, the categories in which the solvers participated (u: Unweighted MaxSAT, w: Weighted MaxSAT, p: Partial MaxSAT, wp: Weighted Partial MaxSAT) and the name of the author(s).

solver	category	author(s)
akmaxsat [Kue10]	u,w,p,wp	A. Kuegel
akmaxsat_ls [Kue10]	u,w,p,wp	
SAT4J-Maxsat [Ber]	u,w,p,wp	D. Le Berre
WPM1 [ABL09]	u,w,p,wp	C. Ansótegui
PM2 [ABL09]	u,p	M.L. Bonet
WPM2 [ABL10]	w, wp	J. Levy
QMaxSat	p	M. Koshimura , T. Zhang
IncMaxSatz [LSL08]	u,w,p,wp	H. Lin , K. Su, C.M. Li
IncWMaxSatz [LSL08]	u,w,p,wp	H. Lin , K. Su, C.M. Li
		J. Argelich
Maxsat_Power	u,w,p,wp	A. Bahrami
LS_Power	u,p	S.R. Mousavi
WMaxsat_Power	u,w,p,wp	M. Farshchian
LSW_Power	w,wp	
wbo 1.4a [MMSP09]	u,w,p,wp	V. Manquinho
wbo 1.4b [MML10]	w,p,wp	J. Marques-Silva, J. Planes
WMaxSatz-2009 [LMP07a]	u,w,p,wp	C.M. Li, F. Manyà
WMaxSatz+ [LMMP10]	w,p,wp	J. Argelich

Table 3.1: Participating solvers in MaxSAT-2010.

Participating solvers can be classified into two main types: branch-and-bound (BnB) solvers and satisfiability-based (SAT-based) solvers. In the first type, we find 10 solvers: akmaxsat, akmaxsat_ls, IncMaxSatz, IncWMaxSatz, Maxsat_Power, LS_Power, WMaxsat_Power, LSW_Power, WMaxSatz-2009, and WMaxSatz+. In the

second type, we find 7 solvers: SAT4J-Maxsat, WPM1, PM2, WPM2, QMaxSAT, wbo 1.4a, and wbo 1.4b.

The next two sections contain an overview of the main solving techniques implemented in the above solvers. Section 3.3 is devoted to BnB MaxSAT solvers, and Section 3.4 is devoted to SAT-based MaxSAT solvers.

3.3 Branch-and-bound algorithms

Competitive exact MaxSAT solvers —as the ones developed by [AMP03, AMP04, AMP05, AMP08, DDDL07, HL06, HLO07, LHdG08, LMP05, LMP06, LS07, PD07, RG07, SZ04, XZ04, XZ05, ZSM03]— implement variants of the following branch-and-bound (BnB) schema for solving the minimization version of MaxSAT: Given a CNF formula ϕ, BnB explores the search tree that represents the space of all possible assignments for ϕ in a depth-first manner. At every node, BnB compares the upper bound (UB), which is the best solution found so far for a complete assignment, with the lower bound (LB), which is the sum of the number of clauses unsatisfied by the current partial assignment plus an underestimation of the number of clauses that will become unsatisfied if the current partial assignment is completed. If $LB \geq UB$, the algorithm prunes the subtree below the current node and backtracks chronologically to a higher level in the search tree. If $LB < UB$, the algorithm tries to find a better solution by extending the current partial assignment by instantiating one more variable. The solution to MaxSAT is the value that UB takes after exploring the entire search tree.

Figure 3.1 shows the pseudo-code of a basic solver for MaxSAT. We use the following notation:

- `SimplifyFormula`(ϕ) is a procedure that transforms ϕ into an equivalent and simpler instance by applying inference rules.

- `EmptyClauses`(ϕ) is a function that returns the number of empty clauses in ϕ.

Algorithm 3.1: `MaxSAT`(ϕ, UB) : Basic BnB algorithm for MaxSAT

Output: The minimum number of unsatisfied clauses in ϕ

Function `MaxSAT` *(ϕ : CNF formula, UB : upper bound)* **: Natural**

```
φ ← SimplifyFormula(φ)
if φ = ∅ or φ only contains empty clauses then
    return EmptyClauses(φ)
LB ← EmptyClauses(φ) + Underestimation(φ)
if LB ≥ UB then
    return UB
x ← SelectVariable(φ)
UB ← Min(UB, MaxSAT(φ_¬x, UB))
return Min(UB, MaxSAT(φ_x, UB))
```

- `Underestimation`(ϕ,UB) is a function that returns an underestimation of the minimum number of non-empty clauses in ϕ that will become unsatisfied if the current partial assignment is extended to a complete assignment.

- LB is a lower bound. We assume that its initial value is 0.

- UB is an upper bound of the number of unsatisfied clauses in an optimal solution. An elementary initial value for UB is the total number of clauses in the input formula, or the number of clauses unsatisfied by an arbitrary interpretation. Another alternative is to solve the LP relaxation of the ILP formulation of the input instance and take as upper bound the number of unsatisfied clauses in the interpretation obtained by rounding variable y_i, for $1 \leq i \leq n$, to an integer solution in a randomized way by interpreting the values of $y_i \in [0, 1]$ as probabilities (set propositional variable x_i to true with probability y_i, and set propositional variable x_i to false with probability $1 - y_i$). Nevertheless, most of the solvers take as initial upper bound the minimum number of unsatisfied clauses that are detected by executing the input formula in a local search solver during a short period of time.

- `SelectVariable`(ϕ) is a function that returns a variable of ϕ following an heuristic.

- ϕ_x ($\phi_{\neg x}$) is the formula obtained by applying the unit clause rule to ϕ using the literal x ($\neg x$).

State-of-the-art MaxSAT solvers implement the basic algorithm augmented with powerful inference techniques, good quality lower bounds, clever variable selection heuristics, learning of hard clauses, non-chronological backtracking, and efficient data structures.

3.3.1 Improving the lower bound with underestimations

The methods more frequently used to compute lower bounds are based on lower bound UP [LMP05, LMP07a]. The underestimation of UP is the number of disjoint inconsistent subformulas that can be detected with unit propagation. UP works as follows: it applies unit propagation until a contradiction is derived. Then, UP identifies, by inspecting the implication graph, a subset of clauses from which a refutation can be constructed, and tries to identify new contradictions from the remaining clauses. The order in which unit clauses are propagated has a significant impact on the quality of the lower bound [LMP06]. Shen and Zhang [SZ04] defined a lower bound computation method, called LB4, which is similar to UP but restricted to Max-2-SAT instances and using a static variable ordering.

Example 3.7 *Given the MaxSAT instance* $\phi = \{\neg x_1, x_1 \vee x_2, x_1 \vee \neg x_2, x_3 \vee x_4, x_3 \vee \neg x_4, \neg x_3 \vee x_4, \neg x_3 \vee \neg x_4\}$*, we can propagate* $\neg x_1$ *and detect, with unit propagation, the inconsistent subformula* $\{\neg x_1, x_1 \vee x_2, x_1 \vee \neg x_2\}$*, and increase the lower bound by 1.*

Once UP cannot detect more inconsistent subformulas, it can be enhanced with failed literal detection [LMMP10, LMP06] as follows: Given a MaxSAT instance ϕ and a variable x occurring positively and negatively in ϕ, we apply UP to $\phi \wedge \{x\}$ and $\phi \wedge \{\neg x\}$. If UP derives a contradiction from $\phi \wedge \{x\}$ and $\phi \wedge \{\neg x\}$, then the

union of the two inconsistent subsets identified by UP is an inconsistent subset of ϕ. UP enhanced with failed literal detection does not need the occurrence of unit clauses in the input formula for deriving a contradiction. While UP only identifies unit refutations, UP enhanced with failed literal detection identifies non-unit refutations too. Since applying detection of failed literals for every variable is time consuming, it is applied to a reduced number of variables in practice.

Example 3.8 *Given the MaxSAT instance of Example 3.7, once UP detects the inconsistent subformula* $\{\neg x_1, x_1 \vee x_2, x_1 \vee \neg x_2\}$*, it cannot detect any other inconsistent subformula from the remaining clauses* ($\phi' = \{x_3 \vee x_4, x_3 \vee \neg x_4, \neg x_3 \vee x_4, \neg x_3 \vee \neg x_4\}$)*. However, we can detect another inconsistent subformula using failed literal detection as follows: we apply UP to* $\phi' \vee \{x_3\}$ *and to* $\phi' \vee \{\neg x_3\}$*, and then derive a contradiction. The union of the two detected inconsistent subsets is an inconsistens subset of* ϕ *formed by the clauses* $\{x_3 \vee x_4, x_3 \vee \neg x_4, \neg x_3 \vee x_4, \neg x_3 \vee \neg x_4\}$*. Observe that in this case the lower bound is increased by 2.*

Modern MaxSAT solvers like akmaxsat [Kue10], MaxSatz [LMP07a, LMMP10] and MiniMaxsat [HLO07] apply either UP or UP enhanced with failed literal detection. Darras et al. [DDDL07] developed a version of UP in which the computation of the lower bound is made more incremental by saving some of the small size disjoint inconsistent subformulas detected by UP. They avoid to redetect the saved inconsistencies if they remain in subsequent nodes of the proof tree, and are able to solve some types of instances faster. A similar approach is implemented in IncMaxSatz [LSL08].

Another approach for computing underestimations is based on first reducing the MaxSAT instance we want to solve to an instance of another problem, and then solve a relaxation of the obtained instance. For example, two solvers of the 2007 MaxSAT Evaluation, Clone [PD07] and SR(w) [RG07], reduce MaxSAT to the minimum cardinality problem. Since the minimum cardinality problem is NP-hard for a CNF formula ϕ and can be solved in time linear in the size of a deterministic decomposable negation normal form (d-DNNF) compilation of ϕ, Clone and SR(w) solve the minimum cardinality problem of a d-DNNF compilation of a relaxation of ϕ. The

worst-case complexity of a d-DNNF compilation of ϕ is exponential in the treewidth of its constraint graph, and Clone and SR(w) obtain a relaxation of ϕ with bounded treewidth by renaming different occurrences of some variables.

Xing and Zhang [XZ05] reduce the MaxSAT instance to the ILP formulation of the minimization version of MaxSAT (c.f. Section 3.1), and then solve the LP relaxation. An optimal solution of the LP relaxation provides an underestimation of the lower bound because the LP relaxation is less restricted than the ILP formulation. In practice, they apply that lower bound computation method only to nodes containing unit clauses. If each clause in the MaxSAT instance has more than one literal, then $y_i = \frac{1}{2}$ for all $1 \leq i \leq n$ and $z_j = 0$ for all $1 \leq j \leq m$ is an optimal solution of the LP relaxation. In this case, the underestimation is 0. Nevertheless, LP relaxations do not seem to be so competitive as the rest of approaches.

3.3.2 Improving the lower bound with inference

Another approach to improving the quality of the lower bound consists in applying inference rules that transform a MaxSAT instance ϕ into an equivalent but simpler MaxSAT instance ϕ'. In the best case, inference rules produce new empty clauses in ϕ' that allow to increment the lower bound. In contrast with the empty clauses derived when computing underestimations, the empty clauses derived with inference rules do not have to be recomputed at every node of the subtree below the current node, so that the lower bound computation is more incremental.

A MaxSAT inference rule is sound if it transforms an instance ϕ into an equivalent instance ϕ'. It is not sufficient to preserve satisfiability as in SAT; ϕ and ϕ' must have the same number of unsatisfied clauses for every possible assignment. Unfortunately, unit propagation, which is the most powerful inference technique applied in DLL-style SAT solvers, is unsound for MaxSAT as the next example shows: the set of clauses $\{x_1, \neg x_1 \vee x_2, \neg x_1 \vee \neg x_2, \neg x_1 \vee x_3, \neg x_1 \vee \neg x_3\}$ has a minimum of one unsatisfied clause (setting x_1 to false), but performing unit propagation with x_1 leads to a non-optimal assignment falsifying two clauses.

MaxSAT inference rules are also called transformation rules in the literature because the premises of the rule are replaced with the conclusion when a rule is applied. If the conclusion is added to the premises as in SAT, the number of clauses that are falsified by an assignment might increase.

The amount of inference performed by existing BnB MaxSAT solvers at each node of the proof tree is poor compared with the inference performed in DLL-style SAT solvers. The simplest inference enforced, when branching on literal ℓ, is the following: the clauses containing ℓ are removed from the instance and the occurrences of $\bar{\ell}$ are removed from the clauses in which $\bar{\ell}$ appears, but the new unit clauses derived as a consequence of removing the occurrences of $\bar{\ell}$ are not propagated as in unit propagation. That inference is typically enhanced with MaxSAT inference rules as the following one: If ϕ_1=$\{l_1,\ \bar{l}_1 \vee \bar{l}_2,\ l_2\} \cup \phi'$, then ϕ_2=$\{\square,\ l_1 \vee l_2\} \cup \phi'$ is equivalent to ϕ_1. This rule is usually represented as follows:

$$\left\{ \begin{array}{c} l_1 \\ \bar{l}_1 \vee \bar{l}_2 \\ l_2 \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} \square \\ l_1 \vee l_2 \end{array} \right\} \tag{3.1}$$

Notice that the rule detects a contradiction from $l_1, \bar{l}_1 \vee \bar{l}_2, l_2$ and, therefore, replaces these clauses with an empty clause. In addition, the rule adds the clause $l_1 \vee l_2$ to ensure the equivalence between ϕ_1 and ϕ_2. For any assignment I such that $I(l_1) = 0, I(l_2) = 1$, or $I(l_1) = 1, I(l_2) = 0$, or $I(l_1) = 1, I(l_2) = 1$, the number of unsatisfied clauses in $\{l_1, \bar{l}_1 \vee \bar{l}_2, l_2\}$ is 1, but for any assignment such that $I(l_1) = 0, I(l_2) = 0$, the number of unsatisfied clauses is 2. Notice that even when any assignment I such that $I(l_1) = 0, I(l_2) = 0$ is not the best assignment for the subset $\{l_1, \bar{l}_1 \vee \bar{l}_2, l_2\}$, it can be the best for the whole formula. By adding $l_1 \vee l_2$, the rule ensures that the number of unsatisfied clauses in ϕ_1 and ϕ_2 is also the same when $I(l_1) = I(l_2) = 0$.

The set of rules implemented in MaxSatz is formed by the previous rule and the following rules:

$$\left\{ \begin{array}{c} l_1 \\ \bar{l}_1 \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} \square \end{array} \right\} \tag{3.2}$$

$$\left\{ \begin{array}{c} l_1 \vee l_2 \vee \cdots \vee l_k \\ \bar{l}_1 \vee l_2 \vee \cdots \vee l_k \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} l_2 \vee \cdots \vee l_k \end{array} \right\} \tag{3.3}$$

$$\left\{ \begin{array}{c} l_1 \\ \bar{l}_1 \vee l_2 \\ \bar{l}_2 \vee l_3 \\ \cdots \\ \bar{l}_k \vee l_{k+1} \\ \bar{l}_{k+1} \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} \square \\ l_1 \vee \bar{l}_2 \\ l_2 \vee \bar{l}_3 \\ \cdots \\ l_k \vee \bar{l}_{k+1} \end{array} \right\} \tag{3.4}$$

$$\left\{ \begin{array}{c} l_1 \\ \bar{l}_1 \vee l_2 \\ \bar{l}_1 \vee l_3 \\ \bar{l}_2 \vee \bar{l}_3 \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} \square \\ l_1 \vee \bar{l}_2 \vee \bar{l}_3 \\ \bar{l}_1 \vee l_2 \vee l_3 \end{array} \right\} \tag{3.5}$$

$$\left\{ \begin{array}{c} l_1 \\ \bar{l}_1 \vee l_2 \\ \bar{l}_2 \vee l_3 \\ \cdots \\ \bar{l}_k \vee l_{k+1} \\ \bar{l}_{k+1} \vee l_{k+2} \\ \bar{l}_{k+1} \vee l_{k+3} \\ \bar{l}_{k+2} \vee \bar{l}_{k+3} \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} \square \\ l_1 \vee \bar{l}_2 \\ l_2 \vee \bar{l}_3 \\ \cdots \\ l_k \vee \bar{l}_{k+1} \\ l_{k+1} \vee \bar{l}_{k+2} \vee \bar{l}_{k+3} \\ \bar{l}_{k+1} \vee l_{k+2} \vee l_{k+3} \end{array} \right\} \tag{3.6}$$

The previous rules can be easily extended to deal with weighted clauses. We illustrate how to do it with the first rule presented:

$$\left\{\begin{array}{l}(l_1, w_1)\\(\bar{l}_1 \vee \bar{l}_2, w_2)\\(l_2, w_3)\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}(\square, w)\\(l_1 \vee l_2, w)\\(l_1, w_1 - w)\\(\bar{l}_1 \vee \bar{l}_2, w_2 - w)\\(l_2, w_3 - w)\end{array}\right\} \tag{3.7}$$

where $w = \min(w_1, w_2, w_3)$.

A more general inference schema is implemented in MiniMaxsat [HLO07]. It detects a contradiction with unit propagation and identifies an unsatisfiable subset. Then, it creates a refutation for that unsatisfiable subset, and applies the MaxSAT resolution rule defined below if the size of the largest resolvent in the refutation is less than 4.

Independently and in parallel, Bonet et al. [BLM06, BLM07], and Heras and Larrosa [HL06] defined a MaxSAT resolution rule that preserves, for every possible truth assignment, the number of unsatisfied clauses. Figure 3.1 shows the MaxSAT resolution rule. Applying the rule amounts to replacing the premises with the conclusion. The tautologies concluded by the rule are removed, and the repeated literals in a clause are collapsed into one.

Moreover, Bonet et al. [BLM06, BLM07] proved the completeness of MaxSAT resolution: by saturating successively w.r.t. all the variables, one derives as many empty clauses as the minimum number of unsatisfied clauses in the MaxSAT input instance. Saturating w.r.t. a variable consists in applying the MaxSAT resolution rule to clauses containing that variable until every possible application of the inference rule only introduces clauses containing that variable (since tautologies are eliminated). Once a MaxSAT instance is saturated w.r.t. a variable, all the clauses containing that variable are not considered to saturate w.r.t. another variable. We refer to [BLM07] for further technical details and for the weighted version of the rule.

$$
\begin{array}{l}
x \vee a_1 \vee \cdots \vee a_s \\
\overline{x} \vee b_1 \vee \cdots \vee b_t \\
\hline\hline
a_1 \vee \cdots \vee a_s \vee b_1 \vee \cdots \vee b_t \\
x \vee a_1 \vee \cdots \vee a_s \vee \overline{b_1} \\
x \vee a_1 \vee \cdots \vee a_s \vee b_1 \vee \overline{b_2} \\
\cdots \\
x \vee a_1 \vee \cdots \vee a_s \vee b_1 \vee \cdots \vee b_{t-1} \vee \overline{b_t} \\
\overline{x} \vee b_1 \vee \cdots \vee b_t \vee \overline{a_1} \\
\overline{x} \vee b_1 \vee \cdots \vee b_t \vee a_1 \vee \overline{a_2} \\
\cdots \\
\overline{x} \vee b_1 \vee \cdots \vee b_t \vee a_1 \vee \cdots \vee a_{s-1} \vee \overline{a_s}
\end{array}
$$

Figure 3.1: The MaxSAT Resolution Rule

3.4 SAT-based MaxSAT algorithms

SAT-based MaxSAT solvers compute an optimal assignment of a MaxSAT instance ϕ through the resolution of a sequence of SAT instances derived from ϕ by relaxing some soft clauses and adding some cardinality constraints.

The first SAT-based approach was defined by Fu and Malik [FM06]. Given a Partial MaxSAT instance $\phi = \{(C_1, 1), \ldots, (C_m, 1), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$[1] where the cost of an optimal assignment is k_{opt}, the algorithm iteratively solves a sequence of SAT instances $\phi^0, \phi^1, \ldots, \phi^k, \ldots, \phi^{k_{opt}}$ such that ϕ^k is satisfiable iff ϕ has an assignment of cost k. Since k_{opt} is the optimal cost, $\phi^0, \phi^1, \ldots, \phi^k, \ldots, \phi^{k_{opt}-1}$ must be unsatisfiable and $\phi^{k_{opt}}$ must be satisfiable. Such instances are defined as follows: $\phi^0 = \phi$, and $\phi^k = (\phi^{k-1} \setminus \phi_c) \cup \{(C'_1 \vee b_1, 1), \ldots, (C'_s \vee b_s, 1)\} \cup enc(\sum_{i=1}^{s} b_i = 1)$, for $0 < k \leq k_{opt}$, where $\phi_c = \{(C'_1, 1), \ldots, (C'_s, 1)\}$ are the soft clauses appearing in the unsatisfiable core detected when proving that ϕ^{k-1} is unsatisfiable, $\{b_1, \ldots, b_s\}$ are fresh auxiliary

[1] $(C_i, 1)$ denotes a soft clause with weight 1, and (C_j, ∞) denotes a hard clause.

variables, and $enc(\sum_{i=1}^{s} b_i = 1)$ is a valid SAT encoding of the cardinality constraint $\sum_{i=1}^{s} b_i = 1$, which is added as a hard constraint. Notice that the clauses in ϕ_c are relaxed, the cardinality constraint prevents the solver to find the same unsatisfiable core in the next iteration, and the underlying SAT solvers should return an unsatisfiable core on unsatisfiable instances. The algorithm terminates when either the detected unsatisfiable core is formed only by hard clauses (i.e., there is no solution) or ϕ^k becomes satisfiable (i.e., k is an optimal cost). This approach has been extended to Weighted Partial MaxSAT in the solvers WPM1 [ABL09], and wbo 1.4a [MMSP09] and wbo 1.4b [MML10]. The last two solvers use a pseudo-Boolean solver instead of a SAT solver. In this case, $enc(\sum_{i=1}^{s} b_i = 1)$ denotes the Pseudo-Boolean constraint $\sum_{i=1}^{s} b_i = 1$ instead of a CNF formula. In Weighted Partial MaxSAT, the value of k is updated adding the minimum weight of the soft clauses involved in the core. Moreover, every involved soft clause is replaced by two copies: one extended with an additional auxiliary variable, and weight equal to the minimum weight of the core; and an unextended one with weight equal to the original weight minus the minimum weight of the core.

In Fu and Malik's approach, a soft clause appearing in more than one detected core is extended with more than one auxiliary variable, and this may hamper the efficiency of the SAT solver. Another alternative for solving Partia MaxSAT, which consumes exactly one auxiliary variable per soft clause, is to define $\phi^0 = \phi$, and define $\phi^k = \{(C_1 \vee b_1, 1), \ldots, (C_m \vee b_m, 1), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\} \cup enc(\sum_{i=1}^{m} b_i \leq k) \cup \bigcup_{r=1}^{k} enc(\sum_{C_i \in B_r} b_i \geq k_r)$, for $0 < k \leq k_{opt}$, where B_i, $1 \leq i \leq k$, is the sequence of soft clauses in the unsatisfiable core detected when solving ϕ^{i-1}, and k_j is the number of subformulas from $B_1, \ldots, B_j$ which are contained in B_j. This solution is implemented in the solver PM2 [ABL09]. Moreover, the weighted version of PM2, WPM2 [ABL10], avoids to maintain two copies of every soft clause appearing in a detected core.

The previous SAT-based MaxSAT solvers search from $k = 0$ to k_{opt} (increasing k while ϕ^k is unsatisfiable). They make a linear search on the lower bounds of the

optimal cost. The solvers SAT4J-Maxsat [Ber] and QMaxSat, which make a linear search on the upper bounds of the optimal cost, start with $k = \sum_{i=1}^{m} w_i$ and decrease this value until the SAT solver reports unsatisfiable. The latest model found is an optimal solution, and the latest k is the optimal cost. Now, ϕ^k is defined as follows: $\phi^k = \{(C_1 \vee b_1, w_1), \ldots, (C_m \vee b_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\} \cup enc(\sum_{i=1}^{m} w_i b_i < k)$.

QMaxSat only deals with Partial MaxSAT instances, and uses Minisat [ES03] and the encoding of cardinality constraints defined in [BB03]. SAT4J-Maxsat, which deals with all the MaxSAT formalisms, uses a pseudo-Boolean solver instead of a SAT solver. Whenever the underlying pseudo-Boolean solver returns satisfiable it checks the satisfying assignment and sets the next k equal to the sum of the weights of the soft clauses with auxiliary variable set to true.

We wrote this chapter before celebrating MaxSAT-2011. In the 2011 edition of the MaxSAT Evaluation, the participating solvers were improved versions of the solver we have described. The only remarkable novelty was a MaxSAT solver based on answer set programming, but its performance was, in general, poor compared with the rest of solvers.

3.5 Summary

We have defined MaxSAT and its extensions: Weighted MaxSAT, Partial MaxSAT and Weighted Partial MaxSAT. Then, we have given a brief description of the solvers that participated in MaxSAT-2010. Finally, we have presented an overview of the solving techniques implemented in both branch-and-bound MaxSAT solvers and SAT-based MaxSAT solvers.

Chapter 4

Encoding CSP into SAT

In this chapter, we present existing and new encodings from CSP into SAT. In Section 4.1, we define the CSP problem and, in Section 4.2, encodings of CSP variables into SAT that allow to maintain a one-to-one mapping between CSP models and SAT models. In Section 4.3 and Section 4.4, we present the well-known direct and support encodings, respectively; and prove that the Exactly-One constraint is compulsory in support encodings. In Section 4.5, we present the variant of the direct encoding known as multivalued encoding. In Section 4.6, we present the log encoding. In Section 4.7, we define the variants of the direct and support encodings that incorporate regular literals, and establish the relationship between the sequential encoding of the cardinality constraint and the regular literals. In Section 4.8, we define a new encoding —the minimal support encoding— and its variants, prove its correctness and show that it does not preserve arc consistency through unit propagation. In Section 4.9, we define another new encoding —the interval-based regular encoding— and prove its correctness. Finally, in Section 4.10, we report on an empirical comparison of the defined encodings on realistic instances. Mappings from CSP into SAT have been investigated before by several authors (see, for example, [BHW03, Gav07, GJ96, Gen02, Kas90, Wal00]).

4.1 CSP preliminaries

A *Constraint Satisfaction Problem (CSP) instance* is defined as a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables, $\mathcal{D} = \{d(X_1), \ldots, d(X_n)\}$ is a set of finite domains containing the values that the corresponding variables may take, and $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of constraints. Each constraint $C_i = \langle S_i, R_i \rangle$ is defined as a relation R_i over a subset of variables $S_i = \{X_{i_1}, \ldots, X_{i_k}\}$, called the *constraint scope*. The relation R_i may be represented extensionally as a subset of the Cartesian product $d(X_{i_1}) \times \cdots \times d(X_{i_k})$.

A CSP is *node consistent* iff, for every variable X_i, every value of the domain of X_i is allowed for the unary constraints on X_i. Given a binary constraint with scope $\{X_i, Y_j\}$, the variable X_i is *arc consistent* relative to the variable Y_j iff, for all $a \in d(X_i)$, there exists $b \in d(Y_j)$ such that (a, b) is in the constraint. A binary constraint with scope $\{X_i, Y_j\}$ is *arc consistent* iff the variable X_i is arc consistent relative to the variable Y_j and the variable Y_j is arc consistent relative to the variable X_i. A binary CSP is *arc consistent* iff all its constraints are arc consistent.

An *assignment* v for a CSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is a mapping that assigns to every variable $X_i \in \mathcal{X}$ an element $v(X_i) \in d(X_i)$. An assignment v satisfies a constraint $\langle \{X_{i_1}, \ldots, X_{i_k}\}, R_i \rangle \in C$ iff $\langle v(X_{i_1}), \ldots, v(X_{i_k}) \rangle \in R_i$. An assigment satisfies a CSP if it satisfies all its constraints.

Given a CSP instance P, the *Constraint Satisfaction Problem (CSP)* consists in deciding whether there exists an assignment that satisfies P. In the sequel, we assume that all CSPs are unary and binary; i.e., the cardinality of all the constraint scopes is at most two.

Example 4.1 *Let P be the CSP instance defined by $\langle X, D, C \rangle$, where $X = \{x_1, x_2, x_3, x_4\}$ is the set of variables, $d(x_1) = \{1, 2\}, d(x_2) = \{0, 1\}, d(x_3) = \{0, 1, 2\}, d(x_4) = \{0, 1\}$, and $C = \{\langle \{x_1, x_2\}, x_1 > x_2 \rangle, \langle \{x_1, x_4\}, x_1 = x_4 \rangle, \langle \{x_2, x_4\}, x_2 < x_4 \rangle, \langle \{x_2, x_3\}, x_2 \neq x_3 \rangle\}$ is the set of constraints. An assignment that satisfies P is $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ and $x_4 = 1$.*

4.2 Encoding CSP variables into SAT

CSP instances contain variables whose domain size can be greater than 2 while the variables occurring in SAT instances take either the value 0 or the value 1. Given a CSP variable X over a domain $d(X) = \{i_1, \ldots, i_m\}$, the most frequent way of encoding X into SAT is associating a Boolean variable x_i with each value $i \in d(X)$ in such a way that x_i is true iff $X = i$. Moreover, since CSP assignments assign exactly one value of the domain to each variable, we have to encode that exactly one of the Boolean variables $\{x_{i_1}, \ldots, x_{i_m}\}$ takes the value 1, and the other variables take the value 0. This is the goal of the *Exactly-One* constraint, which allows to maintain a one-to-one mapping between CSP models and SAT models.

In the sequel, we refer to Boolean variables of the form x_i and with intended meaning $X = i$ as *monosigned variables*, refer to monosigned variables and negated monosigned variables as *monosigned literals*, and refer to encodings only formed by monosigned literals as *standard encodings*. We will present encodings based on monosigned literals, but also encodings based on representing assignments of the form $X = i$ using a number of literals which is logarithmic in the domain size [Pre09], as well as encodings based on the so-called regular literals [AM04].

The Exactly-One constraint is commonly expressed as the conjunction of the ALO *(At-Least-One)* constraint, and the AMO *(At-Most-One)* constraint. The ALO constraint states that at least one of the variables is true, and the AMO constraint states that at most one of the variables is true.

In Section 4.2.1 we describe several encodings of the ALO constraint, and in Section 4.2.2 we describe several encodings of the AMO constraint.

4.2.1 SAT encodings of the ALO constraint

We present different encodings of the ALO constraint, also represented by $\geq_1 (x_1, \ldots, x_n)$. The constraint is true iff at least one out of its n input Boolean variables is true. This constraint is a particular case of the cardinality constraint

$\geq_k (x_1, \ldots, x_n)$, which means that at least k variables are true.

Standard encoding of the ALO constraint

The standard encoding of the ALO constraint is formed by the following clause:

$$x_1 \vee \cdots \vee x_n$$

This encoding requires exactly one n-ary clause for every ALO constraint.

Bitwise encoding of the ALO constraint

Given the ALO constraint $\geq_1 (x_1, \ldots, x_n)$, assume that n is a power of 2. We define $k = \log_2 n$ new Boolean variables $\{b_1, \ldots, b_k\}$ in such a way that each possible assignment to the variables in $\{b_1, \ldots, b_k\}$ is associated with one variable of $\{x_1, \ldots, x_n\}$. If $\{I^i(b_1), \ldots, I^i(b_k)\}$ is the assignment associated with x_i, then the set of literals associated with x_i is $\{l_1^i, \ldots, l_k^i\}$ where, for $1 \leq j \leq k$, it holds that $l_j^i = b_j$ if $I^i(b_j) = 1$, and $l_j^i = \neg b_j$ if $I^i(b_j) = 0$.

In order to have a one-to-one mapping between assignments and variables, if n is not a power of 2, then $k = \lceil \log_2 n \rceil$[1] and, for each assignment $\{I^*(b_1), \ldots, I^*(b_k)\}$ not associated with a variable, a prohibited-value clause $\neg l_1^* \vee \cdots \vee \neg l_k^*$ is added, where, for $1 \leq j \leq k$, it holds that $l_j^* = b_j$ if $I^*(b_j) = 1$, and $l_j^* = \neg b_j$ if $I^*(b_j) = 0$.

The bitwise encoding of $\geq_1 (x_1, \ldots, x_n)$ is formed by the prohibited-value clauses (if any), and the following clauses:

$$\bigwedge_{i=1}^{n} \left(\neg l_1^i \vee \cdots \vee \neg l_k^i \vee x_i \right)$$

This set of clauses is a valid encoding for the ALO constraint: Since the encoding represents all the possible assignments of the k auxiliary variables, there is exactly one disjunction of the form $\neg l_1^i \vee \cdots \vee \neg l_k^i$ which is falsified by any assignment. Any satisfying assignment cannot falsify a prohibited-value clause. Therefore, there will

[1] We write $\lceil x \rceil$ to denote the smallest integer that is greater than or equal to x.

be one clause of the form $\neg l_1^i \vee \cdots \vee \neg l_k^i \vee x_i$ in which $\neg l_1^i \vee \cdots \vee \neg l_k^i$ is falsified and, in order to have a satisfied clause, x_i must be true.

The number of clauses required by this encoding is in $\mathcal{O}(n)$, and the size of the clauses is $\lceil \log_2 n \rceil + 1$, except for the prohibited-value clauses, which have size $\lceil \log_2 n \rceil$.

Example 4.2 *The bitwise encoding of* $\geq_1 (x_1, x_2, x_3)$ *is defined as follows:*

- *As* $n = 3$*, two auxiliary variables* (b_2, b_1) *are needed.*

x_i	b_2	b_1	*literals associated with* x_i	
x_1	0	0	$\neg b_2$	$\neg b_1$
x_2	0	1	$\neg b_2$	b_1
x_3	1	0	b_2	$\neg b_1$
x_4	1	1	b_2	b_1

- *The bitwise encoding clauses are:*

$$(b_2 \vee b_1 \vee x_1)$$
$$(b_2 \vee \neg b_1 \vee x_2)$$
$$(\neg b_2 \vee b_1 \vee x_3)$$
prohibited-value clause
$$(\neg b_2 \vee \neg b_1)$$

4.2.2 SAT encodings of the AMO constraint

We survey several SAT encodings of the AMO constraint, also represented by $\leq_1 (x_1, \ldots, x_n)$, which is true iff at most one out of its n input Boolean variables is true. This constraint is a particular case of the Boolean cardinality constraint $\leq_k (x_1, \ldots, x_n)$.

The pair-wise encoding of the AMO constraint

The most well-known encoding for the constraint $\leq_1 (x_1, \ldots, x_n)$ is the pair-wise encoding, also called naïve encoding in the literature. The encoding is formed by the following clauses:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg x_i \vee \neg x_j$$

This encoding requieres $n\ (n-1)/2$ binary clauses, and does not need to introduce auxiliary variables.

Example 4.3 *The pair-wise encoding of* $\leq_1 (x_1, x_2, x_3, x_4)$ *is defined as follows:*

$$(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_4)$$

The sequential encoding of the AMO constraint

The sequential encoding of the $\leq_1 (x_1, \ldots, x_n)$ was introduced by [Sin05], and is based on a sequential counter circuit, that consists in sequentially counting the number of x_i that are true. The clauses of the encoding are:

$$(\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}))$$

where s_i , $1 \leq i \leq n-1$, are auxiliary variables.

This encoding requieres $3n-4$ binary clauses and $n-1$ auxiliary variables.

Example 4.4 *The sequential encoding of the* $\leq_1 (x_1, x_2, x_3, x_4)$ *is defined as follows:*

- *As* $n = 4$*, three auxiliary variables* s_i *are needed.*
- *The sequential encoding is the following CNF formula* ϕ*:*

$$(\neg x_1 \vee s_1) \wedge (\neg x_4 \vee s_3) \wedge (\neg x_2 \vee s_2) \wedge (\neg s_1 \vee s_2)$$
$$\wedge (\neg x_2 \vee \neg s_1) \wedge (\neg x_3 \vee s_3) \wedge (\neg s_2 \vee s_3) \wedge (\neg x_3 \vee \neg s_2)$$

The bitwise encoding of the AMO constraint

The bitwise encoding of $\leq_1 (x_1, \ldots, x_n)$ uses $k = \lceil \log_2 n \rceil$ auxiliary variables as the bitwise encoding of the ALO constraint [FP01]. It is formed by the prohibited-value clauses (if any), and the following clauses:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil \log_2 n \rceil} \left(\neg x_i \vee l_j^i \right)$$

This is a valid encoding of the AMO constraint, because there is at most one set of literals associated with a variable x_i that is satisfied by an arbitrary assignment. Therefore, at least $n-1$ variables of the form x_i must be set to false. The prohibited-value clause set is needed when the number of variables is not a power of two.

This encoding requieres $n\lceil \log_2 n \rceil$ binary clauses and $\lceil \log_2 n \rceil$ new auxiliary variables.

Example 4.5 *The bitwise encoding of the* $\leq_1 (x_1, x_2, x_3, x_4)$ *is defined as follows:*

- *As* $n = 4$*, two auxiliary variables* (b_2, b_1) *are needed.*

x_i	b_2	b_1	*literals associated with* x_i	
x_1	0	0	$\neg b_2$	$\neg b_1$
x_2	0	1	$\neg b_2$	b_1
x_3	1	0	b_2	$\neg b_1$
x_4	1	1	b_2	b_1

- *The bitwise encoding clauses are:*

$$(\neg x_1 \vee \neg b_2) \wedge (\neg x_1 \vee \neg b_1)$$
$$(\neg x_2 \vee \neg b_2) \wedge (\neg x_2 \vee b_1)$$
$$(\neg x_3 \vee b_2) \wedge (\neg x_3 \vee \neg b_1)$$
$$(\neg x_4 \vee b_2) \wedge (\neg x_4 \vee b_1)$$

4.3 Direct encoding

The direct encoding is probably the most popular encoding from CSP into SAT. The idea of this encoding is to encode into clauses the conflicts among value assignments in constraints. The standard direct encoding[2] consists of the ALO and AMO clauses that ensure that every CSP variable takes exactly one value of its domain. Moreover, for each binary constraint with scope $\{X, Y\}$, there is a binary clause for every nogood. Such clauses are called *conflict* clauses. For example, if $X = 2$ and $Y = 1$ is not allowed, then the conflict clause $\neg x_2 \vee \neg y_1$ is added.

Example 4.6 *The standard direct encoding for the CSP* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle = \langle \{X, Y\}, \{d(X) = \{1, 2, 3\}, d(Y) = \{1, 2, 3\}\}, \{X \leq Y\} \rangle$ *is as follows:*

ALO	$x_1 \vee x_2 \vee x_3$	$y_1 \vee y_2 \vee y_3$				
AMO	$\neg x_1 \vee \neg x_2$	$\neg x_1 \vee \neg x_3$	$\neg x_2 \vee \neg x_3$	$\neg y_1 \vee \neg y_2$	$\neg y_1 \vee \neg y_3$	$\neg y_2 \vee \neg y_3$
conflict	$\neg x_2 \vee \neg y_1$	$\neg x_3 \vee \neg y_1$	$\neg x_3 \vee \neg y_2$			

4.4 Support encoding

In the standard support encoding[3], the idea is to encode into clauses the *support* for each value of X across a constraint instead of encoding conflicts. The support for a value i of a CSP variable X across a binary constraint with scope $\{X, Y\}$ is the set of values of the variable Y which allow $X = i$. If $v_1, v_2, \ldots, v_k$ are the supporting values of the variable Y for $X = i$, we add the clause $\neg x_i \vee y_{v_1} \vee y_{v_2} \vee \cdots \vee y_{v_k}$. There is one clause for each value of the variable X, and one clause for each value of the variable Y. Such clauses are called support clauses. The support clauses on their own do not provide a correct encoding from CSP into SAT. To complete an encoding using support clauses we need to add the ALO and AMO clauses for each CSP variable to ensure that each CSP variable takes exactly one value of its domain.

[2]The standard direct encoding is known as direct encoding in the literature.

[3]The standard support encodings is known as support encoding in the literature.

Example 4.7 *The standard support encoding for the CSP instance of Example 4.6 is as follows:*

ALO	$x_1 \vee x_2 \vee x_3$	$y_1 \vee y_2 \vee y_3$				
AMO	$\neg x_1 \vee \neg x_2$	$\neg x_1 \vee \neg x_3$	$\neg x_2 \vee \neg x_3$	$\neg y_1 \vee \neg y_2$	$\neg y_1 \vee \neg y_3$	$\neg y_2 \vee \neg y_3$
support	$\neg x_2 \vee y_2 \vee y_3$	$\neg y_1 \vee x_1$				
	$\neg x_3 \vee y_3$	$\neg y_2 \vee x_1 \vee x_2$				

The support clause for $X = 1$ is missing because it is subsumed by $y_1 \vee y_2 \vee y_3$, and the support clause for $Y = 3$ is missing because it is subsumed by $x_1 \vee x_2 \vee x_3$.

For the support encoding, ALO and AMO clauses are compulsory. Actually, if AMO clauses are omitted, then the encoding becomes incorrect as we show in the following example.

Example 4.8 *Given the CSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle = \langle \{X, Y\}, \{d(X) = \{1, 2, 3\}, d(Y) = \{1, 2, 3\}\}, \{X = Y\} \rangle$, the standard support encoding without AMO clauses is as follows:*

ALO	$x_1 \vee x_2 \vee x_3$	$y_1 \vee y_2 \vee y_3$	
support clauses	$\neg x_1 \vee y_1$	$\neg x_2 \vee y_2$	$\neg x_3 \vee y_3$
	$x_1 \vee \neg y_1$	$x_2 \vee \neg y_2$	$x_3 \vee \neg y_3$

Assume that all the variables are set to true: $x_1 = x_2 = x_3 = y_1 = y_2 = y_3 = true$. In this case, the encoding should be unsatisfiable because there are combinations such as $x_1 = y_3 = true$ which are not permitted. However, the encoding becomes satisfiable when all the variables are set to true because each clause contains at least one positive literal. This counterexample proves the incorrectness of the encoding. Hence, the ALO and AMO clauses are compulsory in support encodings.

4.5 The multivalued encoding

In the literature, the direct encoding has a variant in which the AMO clauses can be omitted. In this case, each CSP variable can take more than one value simultaneously.

As pointed out in [Pre09], AMO clauses can be omitted in the direct encoding from CSP into SAT in such a way that the CSP is satisfiable iff the resulting SAT encoding is satisfiable. This variant is also known as the multivalued encoding, but it has not an unique name and often is also referred as the direct encoding.

Example 4.9 *The multivalued encoding for the CSP instance of Example 4.6 is as follows:*

$$\begin{array}{ll} ALO & x_1 \vee x_2 \vee x_3 \quad y_1 \vee y_2 \vee y_3 \\ conflict & \neg x_2 \vee \neg y_1 \quad \neg x_3 \vee \neg y_1 \quad \neg x_3 \vee \neg y_2 \end{array}$$

4.6 The log encoding

Another alternative to encode a CSP instance into SAT is the log encoding, which uses a base 2 encoding. Given a CSP variable X with domain $d(X)$, the log encoding requires $k_{d(X)} = \lceil \log_2 |d(X)| \rceil$ auxiliary variables to encode each domain value of X, as the bitwise encoding of the ALO and AMO constraints.

In the log encoding, the assignment $X = i$ is encoded by the clause $l^i_1 \wedge \ldots \wedge l^i_{k_{d(X)}}$, where $\{l^i_1, \ldots, l^i_{k_{d(X)}}\}$ is the set of literals associated with $X = i$ (c.f. Section 4.2.1).

The log encoding is formed by the conflict clause set, which encodes the nogoods of the constraints, and the prohibited-value clauses if there are domains whose size is not a power of 2. Moreover, in contrast to the direct and support encoding, neither ALO nor AMO clauses are required.

Given a nogood $X = i$ and $Y = j$, the conflict clause that encodes this nogood is:

$$\neg l^i_1 \vee \ldots \vee \neg l^i_{k_{d(X)}} \vee \neg l^j_1 \vee \ldots \vee \neg l^j_{k_{d(Y)}}$$

An advantage of the log encoding is that it can be easily extended to deal with contraints of arity greater than two.

Example 4.10 *Let us consider a simple graph colouring problem, consisting of two adjacent vertices X and Y, where each vertex can be coloured with three colours $\{0, 1, 2\}$.*

- *CSP* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle = \langle \{X, Y\}, \{d(X) = \{0, 1, 2\}, d(Y) = \{0, 1, 2\}\}, \{X \neq Y\} \rangle$

- *For each CSP variable* X (Y) *two auxiliary variables* $\{x_{b_0}, x_{b_1}\}$ $(\{y_{b_0}, y_{b_1}\})$ *are needed. The sets of literals associated with the possible assignments of* X *are as follows:*

value domain (i)	b_1	b_0	*literals associates with X=i*	
$X = 0$	0	0	$\neg x_{b_1}$	$\neg x_{b_0}$
$X = 1$	0	1	$\neg x_{b_1}$	x_{b_0}
$X = 2$	1	0	x_{b_1}	$\neg x_{b_0}$
$X = 3$	1	1	x_{b_1}	x_{b_0}

It is similiar for the possible assignments of CSP variable Y*.*

The log encoding for that CSP contains the following conflict and prohibited-value clauses:

conflict clauses	
$\neg[X = 0 \wedge Y = 0]$	$x_{b_0} \vee x_{b_1} \vee y_{b_0} \vee y_{b_1}$
$\neg[X = 1 \wedge Y = 1]$	$\neg x_{b_0} \vee x_{b_1} \vee \neg y_{b_0} \vee y_{b_1}$
$\neg[X = 2 \wedge Y = 2]$	$x_{b_0} \vee \neg x_{b_1} \vee y_{b_0} \vee \neg y_{b_1}$
prohibited-value clauses	
$\neg[X = 3]$	$\neg x_{b_0} \vee \neg x_{b_1}$
$\neg[Y = 3]$	$\neg y_{b_0} \vee \neg y_{b_1}$

The conflict clauses rule out any nogood, and the prohibited-value clauses prevent the bit combination representing the value 3*, which does not belong to the domain.*

There exist alternatives to the prohibited-value clauses for encoding the values in excess. As [Pre07] proposed, such values can be excluded adding extra conflict clauses that interpret the values in excess as allowed domains values.

Example 4.11 *The log encoding for the CSP instance of Example 4.10 obtained by adding extra conflict clauses is as follows:*

conflict clauses	
$\neg[X = 0 \wedge Y = 0]$	$x_{b_0} \vee x_{b_1} \vee y_{b_0} \vee y_{b_1}$
$\neg[X = 1 \wedge Y = 1]$	$\neg x_{b_0} \vee x_{b_1} \vee \neg y_{b_0} \vee y_{b_1}$
$\neg[X = 2 \wedge Y = 2]$	$x_{b_0} \vee \neg x_{b_1} \vee y_{b_0} \vee \neg y_{b_1}$
extra conflict clauses	
$\neg[X = 3 \wedge Y = 3]$	$\neg x_{b_0} \vee \neg x_{b_1} \vee \neg y_{b_0} \vee \neg y_{b_1}$
$\neg[X = 0 \wedge Y = 3]$	$x_{b_0} \vee x_{b_1} \vee \neg y_{b_0} \vee \neg y_{b_1}$
$\neg[X = 3 \wedge Y = 0]$	$\neg x_{b_0} \vee \neg x_{b_1} \vee y_{b_0} \vee y_{b_1}$

The extra conflict clauses exclude the domain value in excess. In this example, the value 3 *is interpreted as the value* 0*, since colour* 3 *and colour* 0 *are considered to be the same colour.*

Interestingly, an encoding mixing the log and support encoding, called log-support encoding, was defined in [Gav07].

4.7 Variants of the direct and support encoding

4.7.1 Regular encodings

The space complexity of the ALO and AMO constraints of the standard direct and support encodings is in $\mathcal{O}(nd^2)$, where n is the number of CSP variables and d is the largest domain size. An alternative to reduce that complexity is to encode the ALO and AMO constraints using both monosigned and regular literals [AM04, BHM01], and leave the conflict and support clauses as in the standard direct and support encodings, respectively. To this end, for every CSP variable X and every value $i \in d(X)$, we associate a monosigned variable x_i, and a Boolean variable $x_i^{\geq}$, called *regular variable*, in such a way that $x_i^{\geq}$ is true iff $X \geq i$. We refer to regular variables

and negated regular variables as *regular literals*. Regular encodings contain both monosigned and regular literals.

The regular direct and support encodings replace the ALO and AMO clauses of each CSP variable X with domain size m with the following clauses [AM04]:

$$\begin{array}{ll} x^{\geq}_{m} \rightarrow x^{\geq}_{m-1} & x_1 \leftrightarrow \neg x^{\geq}_{2} \\ x^{\geq}_{m-1} \rightarrow x^{\geq}_{m-2} & x_2 \leftrightarrow x^{\geq}_{2} \wedge \neg x^{\geq}_{3} \\ \cdots\cdots\cdots\cdots & \cdots\cdots\cdots\cdots \\ x^{\geq}_{3} \rightarrow x^{\geq}_{2} & x_i \leftrightarrow x^{\geq}_{i} \wedge \neg x^{\geq}_{i+1} \\ x^{\geq}_{2} \rightarrow x^{\geq}_{1} & \cdots\cdots\cdots\cdots \\ & x_{m-1} \leftrightarrow x^{\geq}_{m-1} \wedge \neg x^{\geq}_{m} \\ & x_m \leftrightarrow x^{\geq}_{m} \end{array} \tag{4.1}$$

The clauses on the left encode the relationship among the different regular literals of a CSP variable while the clauses on the right link monosigned and regular variables. Clauses on the right are also known as channelling constraints. Notice that $x^{\geq}_{2} \rightarrow x^{\geq}_{1}$ can be omitted. Actually, the clauses in (4.1) produce a better complexity encoding of the ALO and AMO conditions. The space complexity of this encoding is in $\mathcal{O}(nd)$ instead of $\mathcal{O}(nd^2)$.

Interestingly, if in the sequential encoding [Sin05] (c.f. Section 4.2.2) of the AMO constraint, we replace simultaneously x_1 with x_m, x_2 with x_{m-1}, ..., x_m with x_1, and s_1 with $x^{\geq}_{m}$, s_2 with $x^{\geq}_{m-1}$, ..., s_{m-1} with $x^{\geq}_{2}$, we get the following encoding of $\leq 1(x_1, \ldots, x_m)$:

$$(\neg x_m \vee x^{\geq}_{m}) \wedge (\neg x_1 \vee \neg x^{\geq}_{2}) \wedge$$
$$\textstyle\bigwedge_{1<i<m} \left((\neg x_i \vee x^{\geq}_{i}) \wedge (\neg x^{\geq}_{i+1} \vee x^{\geq}_{i}) \wedge (\neg x_i \vee \neg x^{\geq}_{i+1}) \right)$$

which is the regular encoding of the ALO and AMO conditions defined in Equation 4.1 without the ternary clauses. Therefore, in a sense, the popular sequential encoding of $\leq 1(x_1, \ldots, x_m)$ reinvented the regular encoding. Moreover, the encoding resulting from eliminating the ternary clauses in the regular encoding is a valid encoding of the AMO condition.

Let us prove that the ternary clauses of the regular encoding provide an alternative encoding of the ALO condition. Given the clauses

$$\begin{aligned}
&\neg x_2^{\geq} \rightarrow x_1 \\
&x_2^{\geq} \wedge \neg x_3^{\geq} \rightarrow x_2 \\
&\cdots\cdots\cdots\cdots \\
&x_i^{\geq} \wedge \neg x_{i+1}^{\geq} \rightarrow x_i \\
&\cdots\cdots\cdots\cdots \\
&x_{m-1}^{\geq} \wedge \neg x_m^{\geq} \rightarrow x_{m-1} \\
&x_m^{\geq} \rightarrow x_m,
\end{aligned} \tag{4.2}$$

we have that if $x_1 = x_2 = \cdots = x_m = false$, we get a contradiction by applying unit propagation. Otherwise, if any variable x_i is true, then we can build a satisfying assignment by setting to true $x_j^{\geq}$ for $j \leq i$ and setting to false $x_k^{\geq}$ for $k > i$. So, these ternary clauses encode the ALO condition.

Another consequence of this insight is that the sequential encoding is the encoding resulting by replacing, in the regular encoding, the regular ALO condition by the standard ALO condition.

Example 4.12 *The regular direct encoding and regular support encoding for the CSP instance of Example 4.6 are obtained by replacing, in the standard encodings of Example 4.6, the ALO and AMO clauses with the following clauses:*

$$\begin{array}{ll}
\neg x_3^{\geq} \vee x_2^{\geq} & \neg y_3^{\geq} \vee y_2^{\geq} \\
x_1 \leftrightarrow \neg x_2^{\geq} & y_1 \leftrightarrow \neg y_2^{\geq} \\
x_2 \leftrightarrow x_2^{\geq} \wedge \neg x_3^{\geq} & y_2 \leftrightarrow y_2^{\geq} \wedge \neg y_3^{\geq} \\
x_3 \leftrightarrow x_3^{\geq} & y_3 \leftrightarrow y_3^{\geq}
\end{array}$$

4.7.2 Full regular encoding

In the regular encodings, we used both monosigned and regular literals. Another option is just to use regular literals for encoding CSP into SAT. We refer to such encodings as full regular encodings. In this case, we omit the channelling constraints,

and keep the clauses that encode the relationship among the regular literals of every CSP variable. Moreover, we replace the monosigned literals occurring in the conflict and support clauses with the regular representation that encodes the channelling constraints. In a sense, full regular encodings instantiate the channelling constraints.

Example 4.13 *The full regular direct encoding for the CSP instance of Example 4.6 contains the following clauses:*

$$\begin{array}{rlll} & \neg x_3^{\geq} \vee x_2^{\geq} & \neg y_3^{\geq} \vee y_2^{\geq} & \\ \textit{conflict} & \neg x_2^{\geq} \vee x_3^{\geq} \vee y_2^{\geq} & \neg x_3^{\geq} \vee y_2^{\geq} & \neg x_3^{\geq} \vee \neg y_2^{\geq} \vee y_3^{\geq} \end{array}$$

and the full regular support encoding contains the following clauses:

$$\begin{array}{rll} \textit{support} & \neg x_2^{\geq} \vee x_3^{\geq} \vee (y_2^{\geq} \wedge \neg y_3^{\geq}) \vee y_3^{\geq} & y_2^{\geq} \vee \neg x_2^{\geq} \\ & \neg x_3^{\geq} \vee y_3^{\geq} & \neg y_2^{\geq} \vee y_3^{\geq} \vee \neg x_2^{\geq} \vee (x_2^{\geq} \wedge \neg x_3^{\geq}) \end{array}$$

Notice that, when distributivity is applied, the clause $\neg x_2^{\geq} \vee x_3^{\geq} \vee (y_2^{\geq} \wedge \neg y_3^{\geq}) \vee y_3^{\geq}$ *becomes* $\neg x_2^{\geq} \vee x_3^{\geq} \vee y_2^{\geq}$*, and* $\neg y_2^{\geq} \vee y_3^{\geq} \vee \neg x_2^{\geq} \vee (x_2^{\geq} \wedge \neg x_3^{\geq})$ *becomes* $\neg y_2^{\geq} \vee y_3^{\geq} \vee \neg x_3^{\geq}$*.*

Regarding the space complexity, the advantage is that we eliminate the channelling constraints, but the disadvantage is that conflict and support clauses become more complex. In the full regular direct encoding, the space complexity is the same: we do not have channelling constraints and have the same number of conflict clauses, but now they can contain up to four regular literals per clause. The situation is worse in the full regular support encoding because the number of support clauses can grow exponentially due to the application of distributivity. For example, if $d(X) = d(Y) = \{1, 2, \ldots, 10\}$, then the support clause $\neg x_1 \vee y_2 \vee y_4 \vee y_6 \vee y_8$ of the standard support encoding produces an exponential growth in the number of support clauses in the full regular support encoding.

4.7.3 Half regular encoding

The half regular encoding was defined in [AM04], and is between the regular encoding and the full regular encoding. As in the regular encodings, it uses both monosigned

and regular literals. It represents the negative literals with regular literals and the positive literals with monosigned literals.

The advantatge of the half regular encoding is that it avoids applying distributivity in the full regular support encodings. The blowup of the full regular support encoding is due to the encoding of positive literals, and they are now represented using monosigned literals instead of regular literals.

Example 4.14 *The half regular direct encoding for the CSP instance of Example 4.6 contains the following clauses:*

$$\begin{array}{lll} \textit{ALO and AMO} & \neg x_3^{\geq} \vee x_2^{\geq} & \neg y_3^{\geq} \vee y_2^{\geq} \\ & x_1 \leftrightarrow \neg x_2^{\geq} & y_1 \leftrightarrow \neg y_2^{\geq} \\ & x_2 \leftrightarrow x_2^{\geq} \wedge \neg x_3^{\geq} & y_2 \leftrightarrow y_2^{\geq} \wedge \neg y_3^{\geq} \\ & x_3 \leftrightarrow x_3^{\geq} & y_3 \leftrightarrow y_3^{\geq} \end{array}$$

$$\textit{conflict} \quad \neg x_2^{\geq} \vee x_3^{\geq} \vee y_2^{\geq} \quad \neg x_3^{\geq} \vee y_2^{\geq} \quad \neg x_3^{\geq} \vee \neg y_2^{\geq} \vee y_3^{\geq}$$

and the half regular support encoding contains the following clauses:

$$\begin{array}{lll} \textit{support} & \neg x_2^{\geq} \vee x_3^{\geq} \vee y_2 \vee y_3 & y_2^{\geq} \vee x_1 \\ & \neg x_3^{\geq} \vee y_3 & \neg y_2^{\geq} \vee y_3^{\geq} \vee x_1 \vee x_2 \end{array}$$

4.8 Minimal support encoding

Our first original encoding from CSP into SAT is a new version of the standard support encoding, which we call standard minimal support encoding. Our definition follows from the observation that the support encoding contains redundant clauses. More precisely, given a binary constraint C_k with scope $\{X, Y\}$, it is enough to add the support clauses either for the values of X or for the values of Y; it is not necessary to add a clause in each direction. Despite of the number of papers dealing with the support encoding, this fact has gone unnoticed so far.

Definition 4.1 *The standard minimal support encoding is like the standard support encoding except for the fact that, for every constraint C_k with scope $\{X, Y\}$, we add*

either the support clauses for all the domain values of the CSP variable X or the support clauses for all the domain values of the CSP variable Y.

Example 4.15 *A standard minimal support encoding for the CSP instance of Example 4.6 contains either the support clauses $\neg x_2 \vee y_2 \vee y_3, \neg x_3 \vee y_3$ or the support clauses $\neg y_1 \vee x_1, \neg y_2 \vee x_1 \vee x_2$.*

Proposition 4.1 *The minimal support encoding is correct.*

Proof We assume, without loss of generality, that we add the support clauses for all the domain values of the CSP variable X for every constraint C_k with scope $\{X, Y\}$. Given a CSP assignment, we construct its corresponding Boolean assignment by setting the variable x_i to true if the CSP assignment assigns the value i to X; otherwise, we set the variable x_i to false. Given a Boolean assignment that satisfies the minimal support encoding of a CSP, we construct its corresponding CSP assignment by assigning to the CSP variable X the value i if x_i is true. Note that there is exactly one x_i for each CSP variable X which is true because the minimal support encoding contains the ALO and AMO clauses. So, it is a valid CSP assignment.

We prove first that if a CSP assignment satisfies all the constraints of a CSP instance, then its corresponding Boolean assignment satisfies its minimal encoding. Since a CSP assignment assigns exactly one value to each CSP variable, the Boolean assignment satisfies the ALO and AMO clauses. For every constraint C_k with scope $\{X, Y\}$, the CSP assignment assigns a value i to X and a value j to Y. Since $(X = i, Y = j)$ is an allowed combination, among the clauses encoding that constraint, there is a clause of the form $\neg x_i \vee y_j \vee \cdots$ which is satisfied by the Boolean encoding because y_j is true. The remaining clauses are also satisfied by the Boolean assignment because they are of the form $\neg x_l \vee \cdots$, where $l \neq i$, and the Boolean assignment assigns the value false to all variables x_l with $l \neq i$.

We prove now that if a Boolean assignment satisfies the minimal support encoding of a CSP instance P, then its corresponding CSP assignment satisfies P. Assume that the CSP assignment does not satisfy P. Therefore, there exists a constraint C_k of P

with scope $\{X, Y\}$ which is violated because the CSP assignment assigns a value i to X and a value j to Y which correspond to a forbidden combination. In this case, there is exactly one support clause of the form $\neg x_i \vee y_{j_1} \vee \cdots \vee y_{j_k}$ among the support clauses encoding C_k which is not satisfied by the Boolean assignment because x_i is true and $y_{j_1} \neq y_j, \ldots, y_{j_k} \neq y_j$. The rest of support clauses encoding C_k are satisfied by the Boolean assignment because it assigns the value false to all variables x_l with $l \neq i$. Therefore, the Boolean assignment falsifies the minimal support encoding. ■

Unlike the support encoding [Gen02, Kas90], the minimal support encoding does not maintain arc consistency through unit propagation (i.e., an encoding of a binary CSP into SAT is said to be arc consistent through unit propagation if, for every partial assignment, unit propagation achieves at least the same propagation power on the SAT encoding as enforcing arc consistency on the binary CSP). Recall that the direct encoding does not maintain arc consistency too.

Proposition 4.2 *The minimal support encoding does not maintain arc consistency through unit propagation.*

Proof We give a counterexample to prove the proposition. Given the CSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X, Y\}, d(X) = d(Y) = \{1, 2, 3\}, \mathcal{C} = \{C_{XY}\} = \{\{(1,1), (2,2), (3,3)\}\}$ with the following minimal support encoding:

ALO	$x_1 \vee x_2 \vee x_3 \quad y_1 \vee y_2 \vee y_3$
AMO	$\neg x_1 \vee \neg x_2 \quad \neg x_1 \vee \neg x_3 \quad \neg x_2 \vee \neg x_3 \quad \neg y_1 \vee \neg y_2 \quad \neg y_1 \vee \neg y_3 \quad \neg y_2 \vee \neg y_3$
support	$\neg x_1 \vee y_1 \quad \neg x_2 \vee y_2 \quad \neg x_3 \vee y_3$

If x_1 is set to false, then $\neg y_1$ is not derived by unit propagation, and the domain of Y is not arc consistent. Observe that if the support clauses are $\neg y_1 \vee x_1, \neg y_2 \vee x_2, \neg y_3 \vee x_3$, then $\neg y_1$ is derived by unit propagation, and the domain of Y becomes arc consistent. However, if y_1 is set to false, then arc consistency is not maintained in the last case. ■

The regular and full regular support encodings also admit a minimal version, called regular minimal support encoding and full regular minimal support encoding,

respectively. Such minimal versions are obtained by adding, for every constraint C_k with scope $\{X, Y\}$, either the support clauses for all the domain values of X or the support clauses for all the domain values of Y.

4.9 Interval-based support encoding

The support clauses of standard encodings are of the form $\neg x_j \vee y_{v_1} \vee y_{v_2} \vee \cdots \vee y_{v_k}$. If we want to represent $y_{v_1} \vee y_{v_2} \vee \cdots \vee y_{v_k}$ using only regular literals, there may be an exponential blowup due to the application of distributivity to $(y^{\geq}_{v_1} \wedge \neg y^{\geq}_{v_1+1}) \vee (y^{\geq}_{v_2} \wedge \neg y^{\geq}_{v_2+1}) \vee \cdots \vee (y^{\geq}_{v_k} \wedge \neg y^{\geq}_{v_k+1})$. Note that $\neg x_j$ may be translated into the disjunction $\neg x^{\geq}_j \vee x^{\geq}_{j+1}$ because it is a negative literal. So, the full regular support encoding and the full regular minimal support encoding are not useful in practice.

A partial solution for avoiding that exponential blowup was given in [AM04], using the so-called half regular encoding: negative literals are represented with regular literals, and positive literals are represented with monosigned literals. This technique increases the number of regular literals when most of the literals have negative polarity. However, it is not useful for support clauses because most of their literals are generally positive.

We propose a new full regular encoding, based on intervals, in which $y_{v_1} \vee y_{v_2} \vee \cdots \vee y_{v_k}$ is entirely represented by regular literals, and the exponential blowup is avoided. In our encoding, for each support clause, we need m clauses having at most four regular literals, where m is bounded by the domain size.

Given a support clause $\neg x_j \vee y_{v_1} \vee y_{v_2} \vee \cdots \vee y_{v_k}$, the idea behind our new encoding is to represent with intervals the supporting values of variable Y for $X = j$, and then encode, using regular literals, that the supporting variable Y has to take a value inside one of the allowed intervals. We illustrate this idea with an example: Assume that the domain of Y is $\{1, 2, \ldots, 10\}$, and that the support clause is $\neg x_2 \vee y_2 \vee y_3 \vee y_6 \vee y_8 \vee y_9$. Then, Y has to take a value in one of the following intervals: $[2, 3], [6, 6]$, and $[8, 9]$.

The interval-based encoding for this clause is as follows:

$$x_2^{\geq} \wedge \neg x_3^{\geq} \rightarrow y_2^{\geq} \qquad x_2^{\geq} \wedge \neg x_3^{\geq} \wedge y_4^{\geq} \rightarrow y_6^{\geq}$$
$$x_2^{\geq} \wedge \neg x_3^{\geq} \wedge y_7^{\geq} \rightarrow y_8^{\geq} \quad x_2^{\geq} \wedge \neg x_3^{\geq} \rightarrow \neg y_{10}^{\geq}$$

Definition 4.2 *The interval-based (minimal) support encoding is the full regular (minimal) support encoding from CSP into SAT but using the interval-based encoding in the support clauses.*

Example 4.16 *An interval-based minimal support encoding for the CSP instance of Example 4.6 is formed by the following clauses:*

$$\begin{array}{rll} & \neg x_3^{\geq} \vee x_2^{\geq} & \neg y_3^{\geq} \vee y_2^{\geq} \\ \textit{support} & \neg x_2^{\geq} \vee x_3^{\geq} \vee y_2^{\geq} & \neg x_3^{\geq} \vee y_3^{\geq} \end{array}$$

We get the interval-based support encoding if we replace the previous support clauses with: $\neg x_2^{\geq} \vee x_3^{\geq} \vee y_2^{\geq}, y_2^{\geq} \vee \neg x_2^{\geq}, \neg x_3^{\geq} \vee y_3^{\geq}, \neg y_2^{\geq} \vee y_3^{\geq} \vee \neg x_3^{\geq}$.

Proposition 4.3 *The interval-based support encoding does not maintain arc consistency through unit propagation.*

Proof We give a counterexample to prove the proposition. Given the CSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X, Y, Z\}, d(X) = d(Y) = d(Z) = \{1, 2, 3\}$, and $\mathcal{C}$ has the constraints $X \neq Y$ and $X = Z$, the interval-based support encoding for this CSP instance is formed by the following clauses:

$$\begin{array}{lll}
\neg x_3^{\geq} \vee x_2^{\geq} & \neg y_3^{\geq} \vee y_2^{\geq} & \neg z_3^{\geq} \vee z_2^{\geq} \\
x_2^{\geq} \vee y_2^{\geq} & x_2^{\geq} \vee \neg z_2^{\geq} & \\
\neg x_2^{\geq} \vee x_3^{\geq} \vee \neg y_2^{\geq} \vee y_3^{\geq} & \neg x_2^{\geq} \vee x_3^{\geq} \vee z_2^{\geq} & \neg x_2^{\geq} \vee x_3^{\geq} \vee \neg z_3^{\geq} \\
\neg x_3^{\geq} \vee \neg y_3^{\geq} & \neg x_3^{\geq} \vee z_3^{\geq} & \\
y_2^{\geq} \vee x_2^{\geq} & z_2^{\geq} \vee \neg x_2^{\geq} & \\
\neg y_2^{\geq} \vee y_3^{\geq} \vee \neg x_2^{\geq} \vee x_3^{\geq} & \neg z_2^{\geq} \vee z_3^{\geq} \vee x_2^{\geq} & \neg z_2^{\geq} \vee z_3^{\geq} \vee \neg x_3^{\geq} \\
\neg y_3^{\geq} \vee \neg x_3^{\geq} & \neg z_3^{\geq} \vee x_3^{\geq} &
\end{array}$$

If $y_2^{\geq}$ and $\neg y_3^{\geq}$ are set to true (i.e., we set $Y = 2$) and unit propagation is applied, then $\neg z_2^{\geq} \vee z_3^{\geq}$ (i.e., $Z \neq 2$) is not derived. Therefore, arc consistency through unit propagation is not maintained. ■

Proposition 4.4 *The interval-based minimal support encoding does not maintain arc consistency through unit propagation.*

Proof Since the interval-based support encoding does not maintain arc consistency through unit propagation and the interval-based minimal support encoding is subsumed by it, the proposition holds. ■

The counterexample used to prove Proposition 4.3 produces the same encoding for both the full regular support encoding and the interval-based support encoding. Therefore, it also proves that the full regular support encoding does not maintain arc consistency through unit propagation.

4.10 Experimental results

This section reports on the empirical investigation conducted to compare the previous encodings. Experiments were performed on a cluster with 160 2 GHz AMD Opteron 248 Processors with 1 GB of memory, and the cutoff time was 30 minutes per instance.

In the minimal encodings of a binary CSP, for each constraint with scope $\{X, Y\}$, we must include the support clauses either for X or for Y. In the sequel, the support clauses for the variable that produces smaller size clauses are included. To this end, we give a score of 16 to unit clauses, a score of 4 to binary clauses and a score of 1 to ternary clauses and, at the end, choose the variable with the higher sum of scores.

In the tables, we denote the number of instances in each category by # (the number of solved instances is displayed in brackets), the minimal support encoding by `supc`, the regular minimal support encoding by `reg-supc`, the interval-based minimal support encoding by `int-supc`, the standard support encoding by `supxy`, the regular support encoding by `reg-supxy`, the interval-based support encoding by `int-supxy`,

the standard direct encoding by `dir`, the regular direct encoding by `reg-dir`, and the full regular direct encoding by `freg-dir`.

The experiments were designed to compare the encodings from CSP into SAT. We used the solvers MiniSat [ES03] (version 2.2.0), PrecoSAT [Bie10b] (version 236), and Satz [LA97a] (version 215.2). The solved benchmarks were the random binary CSP instances and the graph coloring instances used in [AM04] to compare standard and regular encodings.

The binary CSPs instances were obtained with a generator of uniform random binary CSPs[4] —designed and implemented by Frost, Bessière, Dechter and Regin— that implements the so-called model B: in the class $\langle n, d, p_1, p_2 \rangle$ with n variables of domain size d, we choose a random subset of exactly $p_1 n(n-1)/2$ constraints (rounded to the nearest integer), each with exactly $p_2 d^2$ conflicts (rounded to the nearest integer); p_1 may be thought of as the density of the problem and p_2 as the tightness of constraints, and belong to the hard region of the phase transition described in [SD96]. The difficulty of the instances depends on the selected values for n, d, p_1 and p_2.

The graph coloring instances are flat graphs created with Culberson's graph generator [Cul]. Instances are named $< n, p, k >$, where n is the number of vertices, p is the edge density, and k is the number of colors. The parameter settings were designed to sample across the phase transition following the recommendations given by Culberson[5].

Tables 4.1, 4.2 and 4.3 show the results for binary CSPs with Minisat, PrecoSAT and Satz, respectively. Satz's results indicate that, using this solver, the standard support encoding is superior to the direct encoding on binary CSPs, and reveals that the regular support encoding outperforms the standard support encoding. Our new minimal and interval-based encodings are not so competitive for a classical solver as Satz which does not incorporate conflict-driven clause learning. On the other hand, we observe that interval-based encodings are particularly good in more modern solvers

[4] http://www.lirmm.fr/~bessiere/generator.html

[5] http://web.cs.ualberta.ca/ joe/Coloring/Generators/setting.html

such as Minisat and PrecoSAT, which incorporate clause learning, as well as variable selection heuristics based on the activity of variables or literals, and restart strategies. Besides, the minimal support encodings are better than the non-minimal encodings despite the fact that they do not maintain arc consistency through unit propagation. For example, Minisat (PrecoSAT) with the interval-based and interval-based minimal support encodings solves 35 (8) instances more than Minisat (PrecoSAT) with the standard support encoding. Moreover, the interval-based minimal support encoding is always better than the interval-based support encoding. For Minisat and PrecoSAT, all the variants of the direct encoding have a similar behaviour. For Satz, the standard direct encoding is better than the other variants.

Tables 4.4, 4.5 and 4.6 show the results for the graph coloring problem with Minisat, PrecoSAT and Satz, respectively. For Minisat and PrecoSAT, the regular and full regular direct encodings are the best performing, and the best performing support encodings are the interval-based encodings. In this case, we do not observe significant differences between the minimal and non-minimal variants. For Satz, the regular direct encoding is the best performing, solving 83 more instances than the standard direct encoding and 205 more instances than the full regular direct encoding. Among the different variants of the support encoding, the regular support encoding is the best performing, solving 39 more instances than the regular minimal support encoding, 142 more instances than the standard support encoding, and 143 more instances than the interval-based and interval-based minimal support encodings.

The results of the experiments provide evidence that our new encodings are well-suited for modern SAT solvers such as Minisat and PrecoSAT, which incorporate the most recent SAT technology. On the other hand, the results indicate that the introduction of regular literals may produce substantial gains, and that the fact of maintaining arc consistency through unit propagation is not decisive for getting an efficient encoding. Actually, a recent work by Petke and Jeavons [PJ10] shows that CSPs encoded using the direct encoding and solved with a SAT solver with clause learning can achieve levels of local consistency that cannot be achieved with a SAT

solver without clause learning.

4.11 Summary

We have presented an overview of existing encodings from CSP into SAT, and have defined two new encodings: the minimal support encoding, and the interval-based support encoding. The minimal support encoding reduces the number of clauses of the well-known support encoding, although it does not maintain arc consistency through unit propagation. The interval-based support encoding is the only existing support encoding with regular literals that has polynomial size complexity. The experimental investigation provides empirical evidence that the new encodings have a good performance profile on SAT solvers incorporating clause learning, as well as that the use of regular literals produces significant speedups on some classes of combinatorial problems.

Minisat										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
15_25_80_283	100	5.68(100)	4.87(100)	**0.94(100)**	2.11(100)	1.55(100)	1.37(100)	3.68(100)	1.24(100)	1.58(100)
15_30_80_424	100	16.75(100)	16.49(100)	**2.49(100)**	7.51(100)	4.98(100)	4.05(100)	14.79(100)	3.36(100)	4.60(100)
25_15_198_65	100	20.44(100)	26.24(100)	3.28(100)	19.03(100)	15.74(100)	4.21(100)	4.11(100)	**3.18(100)**	3.88(100)
25_20_198_126	100	211.16(100)	287.34(100)	34.65(100)	309.99(100)	262.11(100)	44.44(100)	54.98(100)	**28.22(100)**	44.33(100
35_10_305_23	100	8.90(100)	10.82(100)	1.62(100)	10.87(100)	9.74(100)	1.83(100)	1.69(100)	**1.53(100)**	1.83(100)
35_15_305_60	100	488.16(99)	642.66(90)	95.45(100)	799.84(65)	802.37(76)	118.57(100)	159.75(100)	**63.89(100)**	105.69(100)
40_8_400_12	100	3.96(100)	4.41(100)	0.79(100)	5.21(100)	4.72(100)	0.89(100)	0.82(100)	**0.74(100)**	0.85(100)
45_10_415_22	100	83.20(100)	106.84(100)	15.82(100)	255.82(100)	218.93(100)	16.16(100)	15.60(100)	**11.73(100)**	16.38(100)
70_5_880_3	100	0.79(100)	0.79(100)	0.25(100)	1.72(100)	1.65(100)	0.28(100)	0.24(100)	**0.23(100)**	0.24(100)
Total	900	899	890	900	865	876	900	900	**900**	900

Table 4.1: Results for random binary CSP with Minisat. Mean time in seconds.

PrecoSAT										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
15_25_80_283	100	2.15(100)	1.90(100)	2.01(100)	1.94(100)	**1.58(100)**	3.53(100)	2.23(100)	4.01(100)	3.44(100)
15_30_80_424	100	6.51(100)	4.69(100)	6.42(100)	5.09(100)	**3.72(100)**	10.48(100)	6.20(100)	11.92(100)	10.83(100)
25_15_198_65	100	16.66(100)	11.48(100)	9.44(100)	16.03(100)	13.81(100)	14.34(100)	**7.66(100)**	12.18(100)	11.55(100)
25_20_198_126	100	300.16(100)	161.95(100)	**113.11(100)**	202.98(100)	152.56(100)	171.24(100)	114.90(100)	146.54(100)	144.13(100)
35_10_305_23	100	8.84(100)	7.58(100)	**4.23(100)**	11.16(100)	9.75(100)	5.91(100)	4.27(100)	4.94(100)	4.81(100)
35_15_305_60	100	779.91(77)	604.71(96)	**317.52(100)**	679.41(92)	607.03(97)	413.92(100)	384.59(100)	377.43(100)	357.87(100)
40_8_400_12	100	4.15(100)	3.54(100)	**1.87(100)**	5.83(100)	5.36(100)	2.59(100)	1.95(100)	2.12(100)	2.10(100)
45_10_415_22	100	138.39(100)	101.10(100)	**46.52(100)**	142.66(100)	128.22(100)	59.03(100)	51.36(100)	51.04(100)	49.85(100)
70_5_880_3	100	0.65(100)	0.65(100)	**0.58(100)**	1.13(100)	1.18(100)	0.63(100)	0.60(100)	0.60(100)	0.59(100)
Total	900	877	896	**900**	892	897	900	900	900	900

Table 4.2: Results for random binary CSP with PrecoSAT. Mean time in seconds.

Satz										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
15_25_80_283	100	7.73(100)	14.61(100)	17.72(100)	17.08(100)	**4.64(100)**	47.38(100)	4.93(100)	543.37(100)	280.74(100)
15_30_80_424	100	19.92(100)	62.18(100)	80.05(100)	56.50(100)	**15.42(100)**	217.90(100)	15.52(100)	872.78(31)	1021.26(42)
25_15_198_65	100	29.09(100)	44.71(100)	76.54(100)	49.05(100)	**14.59(100)**	143.86(100)	22.87(100)	51.59(100)	220.44(100)
25_20_198_126	100	351.60(100)	685.56(71)	1140.32(28)	801.47(100)	**141.36(100)**	285.50(6)	256.62(100)	917.80(38)	1156.24(4)
35_10_305_23	100	15.09(100)	19.64(100)	14.74(100)	10.78(100)	**7.88(100)**	18.35(100)	13.22(100)	15.94(100)	21.72(100)
35_15_305_60	100	705.73(97)	930.65(57)	1078.95(22)	758.17(100)	**269.47(100)**	1168.98(3)	857.76(93)	1241.36(36)	475.48(2)
40_8_400_12	100	7.45(100)	7.46(100)	**3.81(100)**	4.21(100)	3.83(100)	4.55(100)	4.33(100)	6.41(100)	4.43(100)
45_10_415_22	100	194.16(100)	285.21(100)	148.95(100)	**47.31(100)**	54.04(100)	177.19(100)	158.65(100)	151.84(100)	220.00(100)
70_5_880_3	100	10.28(100)	2.13(100)	0.50(100)	0.56(100)	0.95(100)	0.66(100)	**0.38(100)**	1.38(100)	0.40(100)
Total	900	897	828	750	900	**900**	709	893	705	648

Table 4.3: Results for random binary CSP with SATz. Mean time in seconds.

Minisat										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
200_0.13_5	100	65.93(99)	129.77(100)	0.62(100)	4.28(100)	4.67(100)	0.76(100)	0.82(100)	0.77(100)	**0.58(100)**
400_0.02_3	100	0.02(100)	0.03(100)	0.02(100)	0.03(100)	0.04(100)	0.03(100)	0.02(100)	0.03(100)	**0.01(100)**
50_0.5_8	100	615.48(32)	551.00(30)	265.89(84)	528.85(44)	374.15(36)	266.03(84)	407.16(76)	**370.38(86)**	265.81(84)
60_0.5_8	100	695.31(23)	687.51(40)	51.22(98)	430.09(89)	383.92(78)	68.78(99)	**135.38(100)**	56.05(99)	68.61(99)
70_0.5_8	100	738.96(36)	573.33(54)	2.61(100)	133.49(100)	138.05(100)	2.75(100)	19.89(100)	7.79(100)	**2.56(100)**
80_0.5_13	100	63.54(99)	65.85(97)	23.78(98)	48.12(95)	47.08(95)	24.47(98)	21.58(100)	**18.54(100)**	23.71(98)
Total	600	389	421	580	528	509	581	576	**585**	581

Table 4.4: Results for graph coloring with Minisat. Mean time in seconds.

PrecoSAT										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
200_0.13_5	100	2.53(100)	2.74(100)	1.32(100)	4.74(100)	4.50(100)	2.16(100)	2.09(100)	1.38(100)	**1.27(100)**
400_0.02_3	100	0.08(100)	0.09(100)	0.07(100)	0.16(100)	0.18(100)	0.14(100)	0.04(100)	0.05(100)	**0.03(100)**
50_0.5_8	100	487.05(43)	549.66(48)	221.10(81)	469.24(36)	504.18(47)	229.67(83)	512.30(45)	**264.39(85)**	252.72(83)
60_0.5_8	100	389.82(87)	364.34(89)	35.39(98)	345.88(72)	404.09(93)	**30.04(98)**	349.17(87)	36.92(97)	41.93(98)
70_0.5_8	100	59.35(100)	78.72(100)	3.82(100)	116.21(100)	101.01(100)	6.89(100)	59.32(100)	3.95(100)	**3.53(100)**
80_0.5_13	100	91.34(97)	60.52(98)	25.27(98)	90.73(97)	87.14(97)	16.88(95)	44.34(98)	9.30(97)	**12.05(98)**
Total	600	527	535	577	505	537	576	530	579	**579**

Table 4.5: Results for graph coloring with PrecoSAT. Mean time in seconds.

Satz										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
200_0.13_5	100	1382.31(2)	239.03(99)	0.00(0)	0.00(0)	116.73(100)	0.00(0)	**17.69(100)**	51.37(100)	0.00(0)
400_0.02_3	100	0.05(100)	0.05(100)	0.04(100)	0.08(100)	0.07(100)	0.07(100)	**0.02(100)**	0.04(100)	**0.02(100)**
50_0.5_8	100	0.00(0)	1665.86(1)	0.00(0)	0.00(0)	1219.39(7)	0.00(0)	**1008.81(12)**	1277.58(8)	0.00(0)
60_0.5_8	100	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	**956.68(3)**	0.00(0)	0.00(0)
70_0.5_8	100	0.00(0)	1541.16(2)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	**1536.46(4)**	1497.51(2)	0.00(0)
80_0.5_13	100	0.00(0)	744.96(2)	0.00(0)	39.07(1)	591.30(36)	0.00(0)	395.70(3)	**67.48(95)**	0.00(0)
Total	600	102	204	100	101	243	100	222	**305**	100

Table 4.6: Results for graph coloring with Satz. Mean time in seconds.

Chapter 5

Encoding MaxCSP into Partial MaxSAT

In this chapter, we define original encodings from MaxCSP into Partial MaxSAT that extend a number of variants of the direct encoding and (minimal) support encoding from CSP into SAT in such a way that, given a MaxCSP instance P, they produce a Partial MaxSAT instance ϕ in which the minimum number of violated constraints in P is the minimum number of falsified clauses in ϕ. In Section 5.2, we define direct encodings and prove their correctness. In Section 5.3, we define minimal support encodings and prove their correctness. In Section 5.4, we define support encodings, prove their correctness, and prove that a BnB MaxSAT solver does not need to branch on the auxiliary variables of the encoding. In Section 5.5, we define interval-based support encodings. In Section 5.6, we report on an empirical comparison, on realistic instances, of all the defined encoding from MaxCSP into Partial MaxSAT.

As said above, our encodings from MaxCSP into Partial MaxSAT verify that the minimum number of violated constraints in the MaxCSP instance is the same as the minimum number of falsified soft clauses in the Partial MaxSAT instance. Since there is exactly one falsified soft clause for every violated constraint, our encodings can be easily extended to Weighted MaxCSP: Given a constraint C with an associated weight w, we just need to associate the weight w to all the soft clauses encoding C. Interestingly, the planning instances solved in the empirical investigation are Weighted

MaxCSP instances that are encoded as Weighted Partial MaxSAT instances. For the sake of simplicity, we focus our description on the unweighted case in the rest of the chapter, but we would like to remark that our results are also applicable to Weighted MaxCSP.

5.1 MaxCSP preliminaries

The MaxCSP problem is an optimization version of the CSP problem[1]. Given a CSP instance P, the MaxCSP problem consists in finding an assignment that minimizes (maximizes) the number of violated (satisfied) constraints in P.

Example 5.1 *Let P be the CSP instance defined by $\langle X, D, C\rangle$, where $X = \{x_1, x_2, x_3\}$ is the set of variables, $d(x_1) = d(x_2) = d(x_3) = \{1, 2, 3\}$, and $C = \{\langle\{x_1, x_2\}, x_1 < x_2\rangle, \langle\{x_2, x_3\}, x_2 < x_3\rangle, \langle\{x_1, x_3\}, x_1 \neq x_3\rangle\}$ is the set of constraints. An assignment that minimizes the number of violated constraints of P is $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$. The number of violated (satisfied) constraints with this assignment is 1 (2).*

In Weighted MaxCSP, each constraint has an associated weight, and the goal is to minimize the sum of the weights of the violated constraints.

5.2 Direct encodings from MaxCSP into Partial MaxSAT

The standard direct encoding from MaxCSP into Partial MaxSAT is defined by adapting the standard direct encoding from CSP into SAT:

Definition 5.1 *The standard direct encoding of a MaxCSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}\rangle$ is the Partial MaxSAT instance that contains as hard clauses the corresponding ALO and*

[1]For basic concepts on CSPs we refer the reader to Section 4.1.

AMO clauses for every CSP variable in $\mathcal{X}$, and a soft clause $\neg x_i \vee \neg y_j$ for every nogood $(X = i, Y = j)$ of every constraint of $\mathcal{C}$ with scope $\{X, Y\}$.

Example 5.2 *The standard direct encoding for the MaxCSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle = \langle \{X, Y\}, \{d(X) = \{1, 2, 3\}, d(Y) = \{1, 2, 3\}\}, \{X \leq Y\} \rangle$ is as follows:*

ALO	$[x_1 \vee x_2 \vee x_3]$ $[y_1 \vee y_2 \vee y_3]$
AMO	$[\neg x_1 \vee \neg x_2]$ $[\neg x_1 \vee \neg x_3]$ $[\neg x_2 \vee \neg x_3]$ $[\neg y_1 \vee \neg y_2]$ $[\neg y_1 \vee \neg y_3]$ $[\neg y_2 \vee \neg y_3]$
conflict	$(\neg x_2 \vee \neg y_1)$ $(\neg x_3 \vee \neg y_1)$ $(\neg x_3 \vee \neg y_2)$

Proposition 5.1 *Solving a MaxCSP instance is equivalent to solving the Partial MaxSAT problem of its standard direct encoding.*

Proof The hard clauses ensure that exactly one of the Boolean variables that encode a CSP variable is true, and the rest are false in any feasible assignment. Therefore, there is a one-to-one mapping between the set of CSP assignments and the set of feasible assignments of the Partial MaxSAT instance and, moreover, at most one of the conflict clauses that encode a certain constraint can be falsified by a feasible assignment. If the CSP assignment satisfies a constraint, then the corresponding Boolean assignment also satisfies the conflict clauses that encode that constraint because there is no clause forbidding allowed values. If the CSP assignment violates a constraint, then the corresponding Boolean assignment does not satisfy the conflict clause that encodes the forbidden values of the two variables involved in the constraint, and satisfies the remaining clauses. ■

The direct encoding can be easily extended to constraints of higher arity. Given a nogood $(X_1 = i_1, \ldots, X_m = i_m)$, we should add the soft clause $\neg x_{i_1} \vee \ldots \vee \neg x_{i_m}$.

There are other options for defining the direct encoding which amount to introducing auxiliary variables. For example, we could add all the clauses representing nogoods as hard clauses by adding an auxiliary literal c_i to every clause encoding a nogood of every constraint $C_i \in \mathcal{C}$, and adding the unit clause $\neg c_i$ as a soft clause.

Nevertheless, we do not consider this encoding because we realized that its performance profile is worse than the performance profile of the direct encoding (at least for the benchmarks considered in our empirical evaluation).

The regular direct encoding and the full regular direct encoding from MaxCSP into Partial MaxSAT are defined as the corresponding encodings from CSP into SAT where the clauses encoding nogoods are soft and the rest of the clauses are hard.

5.3 Minimal support encodings from MaxCSP into Partial MaxSAT

The standard minimal support encoding from MaxCSP into Partial MaxSAT is defined by adapting the standard minimal support encoding from CSP into SAT:

Definition 5.2 *The standard minimal support encoding of a MaxCSP instance* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *is the Partial MaxSAT instance that contains as hard clauses the corresponding ALO and AMO clauses for every CSP variable in* $\mathcal{X}$*, and as soft clauses the support clauses of the standard minimal support encoding from CSP into SAT.*

Example 5.3 *A standard minimal Partial MaxSAT support encoding for the MaxCSP problem of the CSP instance from Example 5.2 is as follows:*

ALO	$[x_1 \vee x_2 \vee x_3]$ $[y_1 \vee y_2 \vee y_3]$
AMO	$[\neg x_1 \vee \neg x_2]$ $[\neg x_1 \vee \neg x_3]$ $[\neg x_2 \vee \neg x_3]$ $[\neg y_1 \vee \neg y_2]$ $[\neg y_1 \vee \neg y_3]$ $[\neg y_2 \vee \neg y_3]$
support	$(\neg x_2 \vee y_2 \vee y_3)$ $(\neg x_3 \vee y_3)$

Proposition 5.2 *Solving a MaxCSP instance is equivalent to solving the Partial MaxSAT problem of its standard minimal support encoding.*

Proof Proposition 4.1 proves that there is one unsatisfied clause for every violated constraint. Since the standard minimal support encoding is correct, and the hard clauses ensure a one-to-one mapping between MaxCSP and feasible Partial MaxSAT

assignments, the optimal solutions of MaxCSP are exactly the same as the optimal solutions of Partial MaxSAT. ■

The regular minimal support encoding from MaxCSP into Partial MaxSAT is defined as the regular minimal support encoding from CSP into SAT where the support clauses are soft and the rest of the clauses are hard. We do not consider the full regular minimal support encoding because it is not useful in practice.

5.4 Support encodings from MaxCSP into Partial MaxSAT

The standard support encoding from MaxCSP into Partial MaxSAT is defined by adapting the standard support encoding from CSP into SAT:

Definition 5.3 *The standard support encoding of a MaxCSP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is the Partial MaxSAT instance that contains as hard clauses the corresponding ALO and AMO clauses for every CSP variable in $\mathcal{X}$, and contains, for every constraint $C_k \in \mathcal{C}$ with scope $\{X, Y\}$, a soft clause of the form $S_{X=j} \vee c_k$, where c_k is an auxiliary variable, for every support clause $S_{X=j}$ used to encode the support for every value j of X, and contains a soft clause of the form $S_{Y=m} \vee \neg c_k$ for every support clause $S_{Y=m}$ used to encode the support for every value m of Y.*

We introduce an auxiliary variable for every constraint because otherwise there are two unsatisfied soft clauses for every violated constraint of the MaxCSP instance. It is particularly important to have one unsatisfied clause for every violated constraint when mapping weighted MaxCSP instances into weighted MaxSAT instances. In this case, all the clauses encoding a certain constraint have the weight that is associated to that constraint. When a constraint is violated with weight w, this guarantees that there is exactly one unsatisfied clause with weight w.

Example 5.4 *The Partial MaxSAT support encoding for the MaxCSP instance from Example 5.2 is as follows:*

$$
\begin{array}{ll}
ALO & [x_1 \vee x_2 \vee x_3] \;\; [y_1 \vee y_2 \vee y_3] \\
AMO & [\neg x_1 \vee \neg x_2] \;\; [\neg x_1 \vee \neg x_3] \;\; [\neg x_2 \vee \neg x_3] \\
 & [\neg y_1 \vee \neg y_2] \;\; [\neg y_1 \vee \neg y_3] \;\; [\neg y_2 \vee \neg y_3] \\
support & (\neg x_2 \vee y_2 \vee y_3 \vee c_1) \;\; (\neg y_1 \vee x_1 \vee \neg c_1) \\
 & (\neg x_3 \vee y_3 \vee c_1) \qquad (\neg y_2 \vee x_1 \vee x_2 \vee \neg c_1)
\end{array}
$$

Proposition 5.3 *Solving a MaxCSP instance is equivalent to solving the Partial MaxSAT problem of its standard support encoding.*

Proof By introducing auxiliary variables we ensure that the optimal solutions of MaxCSP are exactly the same as the optimal solutions of Partial MaxSAT. The auxiliary variables allow to violate exactly one clause for every violated constraint. ■

In the following proposition we assume that Partial MaxSAT solvers incorporate the rule that replaces any two complementary unit clauses with an empty clause. Actually, most of the BnB solvers implement such a rule.

Proposition 5.4 *When solving a MaxCSP instance with a Partial MaxSAT BnB solver, using the standard support encoding, it is not necessary to branch on the auxiliary variables.*

Proof For every violated constraint C_k with scope $\{X, Y\}$, there is exactly one unsatisfied support clause of the form $\neg x_i \vee y_{j_1} \vee \cdots \vee y_{j_k}$ and one unsatisfied support clause of the form $\neg y_l \vee x_{m_1} \vee \cdots \vee x_{m_s}$ in the support encoding from CSP into SAT. Therefore, these clauses will produce the derivation of the two complementary unit clauses in the support encoding from MaxCSP into Partial MaxSAT: c_k (from $\neg x_i \vee y_{j_1} \vee \cdots \vee y_{j_k} \vee c_k$) and $\neg c_k$ (from $\neg y_l \vee x_{m_1} \vee \cdots \vee x_{m_s} \vee \neg c_k$). The solver will then derive a contradiction from these two clauses. If C_k is satisfied, both support clauses are satisfied and the fact of adding an extra literal does not affect their satisfaction.

■

On the solved benchmarks we did not observe significant differences between branching including auxiliary variables and branching without including them. So, we only report results for branching including auxiliary variables. However, there may exist differences on other types of instances and solvers.

The regular support encoding from MaxCSP into Partial MaxSAT is the standard support encoding from MaxCSP into Partial MaxSAT but using the regular encoding of the ALO and AMO clauses.

5.5 Interval-based support encodings from MaxCSP into Partial MaxSAT

The interval-based support encoding has exactly one violated clause for each direction of each violated constraint. Therefore, the interval-based minimal support encoding from MaxCSP into Partial MaxSAT does not need auxiliary variables, and the interval-based support encoding needs an auxiliary variable for every constraint.

Definition 5.4 *The interval-based minimal support encoding of a MaxCSP instance* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *is the Partial MaxSAT instance that contains as hard clauses the clauses that link the different regular literals for every CSP variable in* $\mathcal{X}$*, and contains as soft clauses the support clauses of the interval-based minimal support encoding from CSP into SAT.*

Definition 5.5 *The interval-based support encoding of a MaxCSP instance* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *is the Partial MaxSAT instance that contains as hard clauses the clauses that link the different regular literals for every CSP variable in* $\mathcal{X}$*, and contains, for every constraint* $C_k \in \mathcal{C}$ *with scope* $\{X, Y\}$*, a soft clause of the form* $S_{X=j} \vee c_k$*, where* c_k *is an auxiliary variable, for every support clause* $S_{X=j}$ *used to encode the support for every value* j *of* X *in the interval-based support encoding from CSP into SAT, and contains a soft clause of the form* $S_{Y=m} \vee \neg c_k$ *for every support clause* $S_{Y=m}$ *used to*

encode the support for every value m of Y in the interval-based support encoding from CSP into SAT.

Example 5.5 *The Partial MaxSAT interval-based support encoding for the MaxCSP instance from Example 5.2 is as follows:*

$$\begin{array}{lll} & [\neg x_3^{\geq} \vee x_2^{\geq}] & [\neg y_3^{\geq} \vee y_2^{\geq}] \\ \textit{support} & (\neg x_2^{\geq} \vee x_3^{\geq} \vee y_2^{\geq} \vee c_1) & (y_2^{\geq} \vee \neg x_2^{\geq} \vee \neg c_1) \\ & (\neg x_3^{\geq} \vee y_3^{\geq} \vee c_1) & (\neg y_2^{\geq} \vee y_3^{\geq} \vee \neg x_3^{\geq} \vee \neg c_1) \end{array}$$

Note that we get a Partial MaxSAT interval-based minimal support encoding replacing the soft clauses with either $(\neg x_2^{\geq} \vee x_3^{\geq} \vee y_2^{\geq})$, $(\neg x_3^{\geq} \vee y_3^{\geq})$ or $(y_2^{\geq} \vee \neg x_2^{\geq})$, $(\neg y_2^{\geq} \vee y_3^{\geq} \vee \neg x_3^{\geq})$.

5.6 Experimental results

This section reports on the empirical investigation conducted to compare the previous encodings. Experiments were performed on a cluster with 160 2 GHz AMD Opteron 248 Processors with 1 GB of memory, and the cutoff time was 30 minutes per instance.

In the minimal encodings of a binary MaxCSP, for each constraint with scope $\{X, Y\}$, we must include the support clauses either for X or for Y. In the sequel, the support clauses for the variable that produces smaller size clauses are included. To this end, we give a score of 16 to unit clauses, a score of 4 to binary clauses and a score of 1 to ternary clauses and, at the end, choose the variable with the higher sum of scores.

In the tables, we denote the number of instances in each category by # (the number of solved instances is displayed in brackets), the minimal support encoding by `supc`, the regular minimal support encoding by `reg-supc`, the interval-based minimal support encoding by `int-supc`, the standard support encoding by `supxy`, the regular support encoding by `reg-supxy`, the interval-based support encoding by `int-supxy`, the standard direct encoding by `dir`, the regular direct encoding by `reg-dir`, and the full regular direct encoding by `freg-dir`.

The experiments of this section were designed to compare the defined encodings from MaxCSP into Partial MaxSAT. The solvers used were the last publicly available versions of WMaxSatz [LMMP10, LMP07a], SAT4J-Maxsat [LP10] and MSUnCore [MSP08]. The benchmarks solved were the clique tree instances with different constraint tightness (kbtree) used in [ACLM09a, ACLM09b], and the planning instances from the Soft CSP repository. [2] All these benchmarks are also used in the MaxSAT Evaluation [ALMP08].

Table 5.1, 5.2 and 5.3 show the results for kbtree with WMaxSatz, and planning instances with WMaxSatz and MSUnCore, respectively. For kbtree, we show the results for WMaxSatz and observe that the minimal support encoding is the best performing (174 instances solved) and then the interval-based minimal support encoding (156 instances solved, some of them not solved by any other encoding). The non-minimal version only solves 50 instances. The third best performing encoding is the standard direct encoding. We do not show the results for SAT4J-Maxsat and MSUnCore because there are no significant differences. For planning, we show the results for WMaxSatz and MSUnCore, because SAT4J-Maxsat solves the same number of instances (72) for all the variants except for the regular direct encoding (53). Among the direct encodings, the performance depends on the solver. Among the support encodings, the minimal variants are the best performing ones.

The reported results provide evidence that our new encodings are well-suited for branch and bound MaxSAT solvers such as WMaxSatz, and SAT-based MaxSAT solvers such as MSUnCore and SAT4J-Maxsat. In particular, the use of minimal variants and the introduction of regular literals may produce substantial gains. The interval-based approach depends on the solver and the class of instances to be solved. Moreover, we observe that the standard support encoding is not competitive with the new encodings.

[2] http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/BenchmarkS.

Kbtrees with WMaxSatz. Mean time in seconds										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
w9-s7-h3-d510	50	**0.01(50)**	0.31(50)	**0.01(50)**	27.80(50)	153.75(48)	**0.01(50)**	**0.01(50)**	1.45(50)	**0.01(50)**
w9-s7-h3-d520	50	52.66(50)	61.76(50)	63.59(50)	0.00(0)	0.00(0)	0.00(0)	**2.35(50)**	296.08(48)	164.08(50)
w9-s7-h3-d530	50	412.23(50)	607.75(37)	841.66(31)	0.00(0)	0.00(0)	0.00(0)	**344.13(50)**	1698.04(1)	0.00(0)
w9-s7-h3-d540	50	**1127.21(20)**	1497.01(3)	1529.31(1)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
w9-s7-h3-d550	50	**1177.44(4)**	0.00(0)	585.54(1)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
w9-s7-h3-d560	50	0.00(0)	0.00(0)	**1379.33(5)**	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
w9-s7-h3-d570	50	0.00(0)	0.00(0)	**1643.55(2)**	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
w9-s7-h3-d580	50	0.00(0)	0.00(0)	**1148.98(11)**	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
w9-s7-h3-d590	50	0.00(0)	0.00(0)	**1445.80(5)**	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
Total	450	**174**	140	156	50	48	50	150	99	100

Table 5.1: Results for Kbtrees with WMaxSatz. Mean time in seconds.

Planning with WMaxSatz. Mean time in seconds										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
bwt	11	2.28(9)	4.80(9)	0.01(2)	691.85(5)	0.27(2)	0.05(2)	**1.74(9)**	2.50(9)	0.10(5)
depot	4	**0.01(4)**	**0.01(4)**	**0.01(4)**	0.14(4)	0.28(4)	0.06(4)	**0.01(4)**	**0.01(4)**	**0.01(4)**
driverlog	20	191.65(9)	344.91(9)	0.26(3)	2.95(3)	9.14(3)	0.82(3)	**12.63(15)**	23.40(15)	0.02(3)
driverlogs	2	18.33(2)	28.96(2)	**6.86(2)**	1192.64(1)	0.00(0)	0.00(0)	8.55(2)	24.63(2)	18.22(2)
logistics	4	0.13(4)	0.31(4)	0.11(4)	27.09(4)	135.34(4)	1.60(4)	**0.10(4)**	0.42(4)	0.11(4)
mprime	12	70.97(12)	154.49(12)	0.02(4)	0.13(4)	0.25(4)	0.06(4)	**0.99(12)**	2.32(12)	6.64(8)
rovers	4	0.80(4)	1.94(4)	0.00(0)	1606.91(3)	0.00(0)	0.00(0)	**0.12(4)**	0.31(4)	0.22(4)
satellite	7	2.34(4)	27.25(4)	0.00(0)	0.00(0)	0.00(0)	0.00(0)	**0.05(4)**	0.08(4)	7.87(4)
zenotravel	8	1.81(8)	5.74(8)	0.00(0)	352.07(4)	1130.12(4)	0.00(0)	**0.57(8)**	0.99(8)	0.20(4)
Total	72	56	56	19	28	21	17	**62**	62	38

Table 5.2: Results for planning with WMaxSatz. Mean time in seconds.

Planning with MSUnCore. Mean time in seconds										
Instance set	#	supc	reg-supc	int-supc	supxy	reg-supxy	int-supxy	dir	reg-dir	freg-dir
bwt	11	0.92(11)	0.85(11)	3.34(11)	3.30(11)	10.08(9)	55.08(11)	**0.81(11)**	1.04(9)	1.55(10)
depot	4	0.64(4)	0.04(3)	0.42(4)	**0.15(4)**	0.38(4)	0.24(4)	0.02(3)	0.00(0)	0.03(2)
driverlog	20	0.02(2)	0.15(2)	2.67(3)	0.07(2)	0.12(3)	0.11(2)	**0.03(3)**	0.00(0)	0.02(1)
driverlogs	2	0.00(0)	0.00(0)	7.45(1)	3.35(1)	3.28(1)	0.00(0)	0.00(0)	**0.89(1)**	2.07(1)
logistics	4	0.14(3)	120.03(4)	**0.35(4)**	0.91(4)	1.11(4)	4.64(4)	0.17(2)	0.00(0)	0.30(3)
mprime	12	3.37(11)	4.65(11)	1.32(11)	5.43(11)	3.15(10)	43.76(11)	**0.47(11)**	1.39(7)	1.38(11)
rovers	4	0.83(3)	1.07(2)	0.97(3)	3.41(1)	299.99(2)	0.00(0)	**0.22(3)**	0.25(3)	0.60(3)
satellite	7	0.00(0)	294.46(2)	43.92(2)	60.86(2)	9.34(2)	0.00(0)	0.00(0)	0.00(0)	**3.98(4)**
zenotravel	8	0.32(3)	0.63(3)	0.61(2)	4.16(1)	1.65(3)	66.36(4)	**0.71(4)**	0.18(2)	6.33(4)
Total	72	37	38	**41**	37	38	36	37	22	39

Table 5.3: Results for planning MSUnCore. Mean time in seconds.

5.7 Summary

We have defined a number of original encodings from MaxCSP into Partial MaxSAT, which are the extension of a number of encodings from CSP into SAT. Firstly, we have defined the direct, support and the minimal support encodings from MaxCSP into Partial MaxSAT, and have proved its correctness. Moreover, we have proved that a BnB solver does not need to branch on the auxiliary variables introduced in the support encodings. Secondly, we have extended the interval-based encodings from CSP into SAT, obtaining its Partial MaxSAT version: the interval-based support encoding and the interval-based minimal support encoding. Finally, we have reported on an empirical investigation that we have conducted to compare the performance of all the defined encodings from MaxCSP into Partial MaxSAT.

Chapter 6

Generation of Hard MaxSAT Instances

MaxSAT solvers have made tremendous progress in terms of performance in recent years. Nowadays, we count with fast exact solvers but there has not been parallel progress in the generation of challenging benchmarks for studying the scaling behavior of solvers, and comparing their performance. Most experimental investigations only include, besides random Max$k$SAT instances, the sets of individual instances submitted to the MaxSAT evaluations held so far [ALMP08, ALMP11a, ALMP11c]. The problem with many of the latter instances is that they are becoming easy for modern solvers, and do not allow to analyse the scaling behavior. To cope with that problem, we propose several generators of MaxSAT instances of adjustable hardness.

The first generator produces random (unweighted) MaxSAT instances with unary and binary clauses. It mixes the polynomially solvable problem Max1SAT and the NP-hard problem Max2SAT. The second generator produces random Partial Max2SAT instances with a variable number of soft clauses. Notice that Partial Max2SAT without soft clauses is polynomially solvable, but becomes NP-hard when soft clauses are added. In both cases, in which we consider pure random instances, we show that modern solvers interpolate smoothly from a polynomial optimization problem to one which is NP-hard.

The third and four generators produce structured instances. The third one pro-

duces MaxSAT instances that encode the problem of finding a maximum cut in graphs which have been generated from a bipartite graph by randomly adding a variable number of edges. The fourth one produces Weighted Partial MaxSAT instances that encode a variant of the rectangular bin packing problem, in which the objective is to maximize the number of rectangular pieces of different size which can be placed in a rectangular bin containing obstacles. In both cases, we provide a testbed whose computational difficulty may be adjusted by a parameter.

The chapter is structured as follows. In Section 6.1, we describe the generators we have developed and the used MaxSAT encodings. In Section 6.2, we report on the empirical investigation which allows to assess the computational difficulty of the instances produced by the new generators. In Section 6.3, we present some concluding remarks.

6.1 Problems and generators

We have considered four different problems for creating MaxSAT instance generators: Max1+pSAT, Partial Max2SAT, MaxCut and rectangular bin packing.

The goal of comparing MaxSAT algorithms with different problems is to try to understand some basic characteristics of the instances that affect the performance of the algorithms, so that the results of this analysis can be used to help selecting the most appropriate algorithm for solving other problems with hard and soft constraints.

6.1.1 Max1+pSAT

The first generator produces random (unweighted) MaxSAT instances with unary and binary clauses. It mixes the polynomially solvable problem Max1SAT and the NP-hard problem Max2SAT. It is a generator of random instances for the Max1+pSAT problem [SW02], where a fraction of p clauses are binary and the rest are unary. Both unary and binary clauses are selected uniformly at random from the

set of all possible set of unary and binary clauses, respectively, that can be created from a given set of Boolean variables.

6.1.2 Partial Max2SAT

The second generator produces random Partial Max2SAT instances with a variable number of soft clauses. Each clause is selected uniformly at random from the set of all possible binary clauses that can be created from a given set of Boolean variables. Notice that Partial Max2SAT without soft clauses is polynomially solvable, but becomes NP-hard when soft clauses are added.

6.1.3 MaxCut

The third generator produces MaxSAT instances of the MaxCut problem. The MaxCut problem consists in finding a maximum cut in a graph $G(V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E = \{(v_i, v_j) | v_i, v_j \in V\}$ are the vertex set and the edge set, respectively.

Before explaining the generator, we define some preliminary concepts: let $G = (V, E)$ be an undirected graph. A cut is a partition of the vertices in V into two disjoint subsets S and T. Any edge $(u, v) \in E$ with $u \in S$ and $v \in T$ is said to be crossing the cut, and is a cutting edge. The size of the cut is the number of cutting edges. A maximum cut (MaxCut) is then defined as a cut of G of maximum size.

A bipartite graph G is a graph whose vertices can be partitioned into two disjoint subsets V_1 and V_2 such that every edge connects a vertex in V_1 to one in V_2. They are denoted by $G(V_1 \cup V_2, E)$, where V_1 is the left partition and V_2 is the right partition of the set of vertices, and $E \subseteq V_1 \times V_2$ is the edge set. The MaxCut problem is essentially the same as the problem of finding a bipartite subgraph with the maximum number of edges.

The generator produces MaxSAT intances of the MaxCut problem for graphs with m edges that have been generated as the union of a bipartite graph with $(1 - p) \cdot m$ edges and a randomly selected set of $p \cdot m$ edges, where p is the fraction parameter

that varies from 0 to 1. The bipartite subgraph is generated following the algorithm defined in [ABFM08], in such a way that the graph has a high expansion.

The expansion of a subset X from the vertices of $G(V_1 \cup V_2, E)$ is defined to be the ratio $|N(X)|/|X|$, where $N(X) = \{w \in (V_1 \cup V_2) \backslash X | \exists v \in X, (v, w) \in E\}$ is the set of outside neighbours of X. A set is considered to be high expanding when its expansion is greater than 1, that means that the set of different outside neighbors of X is larger than X, so it is well connected with the rest of the graph.

Observe that, in a bipartite graph, one can always determine an optimal cut in polynomial time by simply finding the bipartition of the graph. However, in our case, as the fraction p of random edges is greater than zero, the whole graph is not necessarily bipartite. Thus, the optimal solution may be different from the bipartition of the bipartite subgraph. Moreover, we use the high expanding bipartite graphs from [ABFM08] because this way we increase the chances to have many sub-optimal partitions, given that every high expanding subset of vertices is connected to a big subset of vertices, so that the cut they define is of big size.

In order to encode MaxCut into MaxSAT, we used the encoding in [Yan94]: Given a graph with m edges, we create, for each edge (v_i, v_j), exactly two binary clauses $(x_i \vee x_j)$ and $(\neg x_i \vee \neg x_j)$. The intended meaning of x_i (x_j) is that it takes the value 1 if v_i (v_j) belongs to the set S of the vertex set partition, and takes the value 0 if v_i (v_j) belongs to the set T of the vertex partition. If ϕ is the collection of such binary clauses, then the MaxCut instance has a cut of size k if, and only if, the MaxSAT instance has an assignment under which $m + k$ clauses are satisfied.

Example 6.1 *Given a graph $G = (V, E)$, with a vertex set $V = \{v_1, v_2, v_3, v_4, v_5\}$ and with an edge set $E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_5), (v_2, v_3), (v_4, v_5), (v_3, v_4)\}$ (see Figure 6.1). The MaxCut problem is encoded as a MaxSAT instance as follows:*

(i) We define a set of propositional variables $\{x_1, x_2, x_3, x_4, x_5\}$; the intended meaning of variable x_i is that vertex $v_i \in S$ if $x_i = 1$ and $v_i \in T$ if $x_i = 0$.

(ii) There are two clauses for every edge $(v_i, v_j) \in E$*:*

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge$$
$$(x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge$$
$$(x_1 \vee x_4) \wedge (\neg x_1 \vee \neg x_4) \wedge$$
$$(x_1 \vee x_5) \wedge (\neg x_1 \vee \neg x_5) \wedge$$
$$(x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge$$
$$(x_4 \vee x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge$$
$$(x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4)$$

An optimal solution for this MaxSAT instance is the assignment $\{I(x_1) = 1, I(x_2) = 0, I(x_3) = 1, I(x_4) = 0, I(x_5) = 1\}$. Notice that, $S = \{v_1, v_3, v_5\}$ and $T = \{v_2, v_4\}$. Since this assignment satisfies 12 clauses and the graph has $m = 7$ edges, the MaxCut instance has a maximum cut of size 5. The double lines of Figure 6.1 represent the cutting edges.

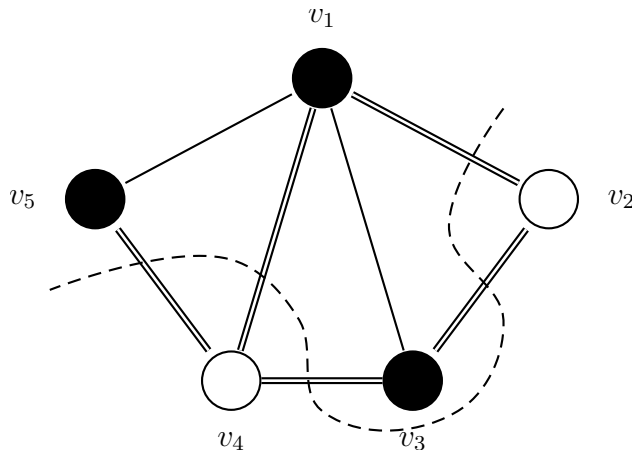

Figure 6.1: A maximum cut of size 5.

6.1.4 Rectangular bin packing

The fourth generator produces Weighted Partial MaxSAT instances of the rectangular bin packing problem. We are given a set of n rectangular pieces of different size which

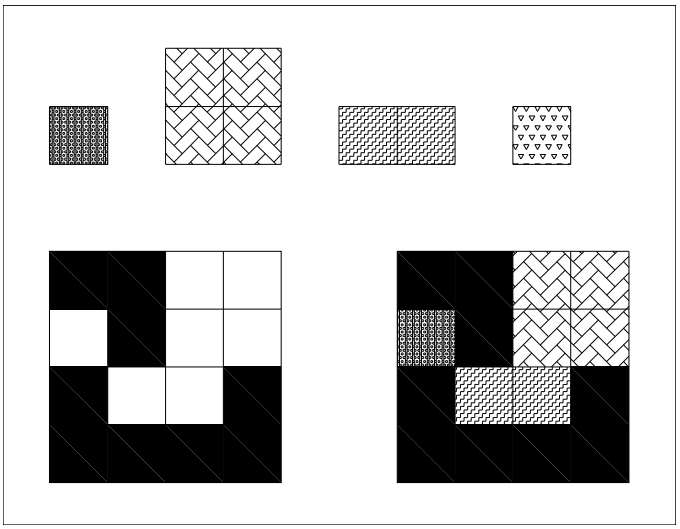

Figure 6.2: Example of bin packing instances

must be placed in a finite rectangular bin of height H and width W. We divide such a bin in $H+1$ rows and $W+1$ columns, taking as origin the position $(0,0)$ (i.e.; rows are named $0,1,\ldots,H$, and columns are named $0,1,\ldots,W$). Each piece i is characterized by its height h_i ($h_i < H$) and width w_i ($w_i < W$), and may not be rotated. We will say that a piece is in position (i,j) if the left upper corner of the piece is in row i and column j. Moreover, we assume that some cells are filled and, therefore, no piece can be located there; in other words, there are some obstacles. The generator has a parameter that allows to adjust the number of cells which are filled; the location of such cells is randomly selected following a uniform distribution. The problem we want to solve is to locate as much rectangular pieces as possible in the bin taking into account the size of the pieces. That is, to maximize the sum of the sizes of the located pieces in the bin. Figure 6.2 shows a bin of height 4 and width 4 containing 9 filled cells representing obstacles (left), and a bin which represents an optimal solution (right). This optimal bin locates 3 pieces, whose sizes are 2×2, 2×1, and 1×1. There is one piece of size 1×1 which cannot be located.

Next, we present the different encodings of the rectangular bin packing produced by the generator. We defined two different encodings, called BP1 and BP2.

Encoding BP1

We define the Weighted Partial MaxSAT encoding BP1 for the rectangular bin packing problem. For each rectangular piece k with height h_k and width w_k, we have the set of propositional variables $\{r_i^k, c_j^k | h_k \leq i \leq H, 0 \leq j \leq W - w_k\} \cup \{x_k\}$. Variable r_i^k (c_j^k) is true if the left upper corner of piece k is located in row i (column j). Variable x_k is true if piece k is located in the bin.

The first block of hard clauses ensures that the left upper corner of piece k in the bin is just in one row. So, for each piece k, we first add the clause:

$$\neg x_k \vee r_{h_k}^k \vee r_{h_k+1}^k \vee \cdots \vee r_H^k \tag{6.1}$$

Since hard clauses must be satisfied in any optimal solution, variable x_k is false if piece k is not in an allowed row.

Then, for every two rows i, j such that $h_k \leq i < j \leq H$, we add the clause:

$$\neg x_k \vee \neg(r_i^k \wedge r_j^k) \tag{6.2}$$

The second block of hard clauses ensures that the left upper corner of piece k in the bin is just in one column. Otherwise, x_k should be false. First, we add the clause

$$\neg x_k \vee c_0^k \vee c_1^k \vee \cdots \vee c_{W-w_k}^k \tag{6.3}$$

Then, for every two columns i, j such that $0 \leq i < j \leq W - w_k$, we add the clause:

$$\neg x_k \vee \neg(c_i^k \wedge c_j^k) \tag{6.4}$$

The third block of hard clauses ensures that pieces do not overlap with the location of the obstacles. For every piece k and every cell (i, j) filled with an obstacle, we add the clauses:

$$\neg r_l^k \vee \neg c_m^k \tag{6.5}$$

where $i \leq l \leq i + h_k - 1$ and $j - w_k + 1 \leq m \leq j$.

The fourth block of hard clauses ensures that any two pieces s and k do not overlap: for every allowed position (i, j) ($h_s \leq i \leq H$ and $0 \leq j \leq W - w_s$) of piece s, we add

the following clauses:

$$\neg(r_i^s \wedge c_j^s) \vee \neg(r_l^k \wedge c_m^k) \tag{6.6}$$

for each allowed position (l, m) ($h_k \leq l \leq H$ and $0 \leq m \leq W - w_k$) of piece k, and where $i - h_s + 1 \leq l \leq i + h_k - 1$ and $j - w_k + 1 \leq m \leq j + w_s - 1$.

Finally, there is a block of soft clauses. If we just would like to maximize the number of pieces in every bin, we should add the unit clauses $\{x_1, \ldots, x_n\}$, where n is the total given number of rectangular pieces, but as we want to take into account the size of the pieces, we add the area of the piece as weight. So, the block of soft clauses is formed by the following weighted unit clauses:

$$(x_1, h_1 \times w_1), \ldots, (x_n, h_n \times w_n) \tag{6.7}$$

Notice that the set of clauses 6.2 ensures that if a piece k is in the bin, it must be located in at most one row, and the set of clauses 6.4 ensures that if a piece k is in the bin, it must be located in at most one column. So, these sets of clauses are encoding the at-most-one (AMO) constraints. If these clauses are removed in the encoding described above, a new valid variant of the encoding is obtained. Removing the AMO clauses is valid because it does not change the value of the optimal solutions. To see the reason, consider an optimal solution found with the variant without AMO clauses that contains a piece located in more than one position. Then, if it would be possible to eliminate any of the repeated positions of that piece to incorporate an additional piece, then the solution could be improved and it would not be an optimal solution. However, it could be the case that the AMO clauses help solvers to rule out these non-improving solutions during search. So, we will consider two variants. Encoding BP1 without AMO clauses (BP1-NAMO) and encoding BP1 with AMO clauses (BP1-AMO).

Encoding BP2

The Weighted Partial MaxSAT encoding BP2 for the rectangular bin packing problem is defined as follows: for each rectangular piece k with height h_k and width w_k, we have

the set of propositional variables $\{c_{ij}^k | h_k \leq i \leq H, 0 \leq j \leq W - w_k\} \cup \{x_k\}$. Variable c_{ij}^k is true if the left upper corner of piece k is located in row i and in column j, i.e., in the cell (i, j). Variable x_k is true if piece k is located in the bin.

The first block of hard clauses ensures that any piece k in the bin is just in one cell. So, for each piece k, we first add the clause:

$$\neg x_k \vee \Big(\bigvee_{(i,j) \in V} c_{ij}^k \Big) \tag{6.8}$$

where $V = \{h_k, \ldots, H\} \times \{0, \ldots, W - w_k\}$. Notice, that this first set of clauses ensures that a piece k must be located in at least one cell. Since hard clauses must be satisfied in any optimal solution, variable x_k is false if piece k is not in an allowed cell.

Then, for every two cells $(i, j), (m, l)$ such that $h_k \leq i \leq m \leq H$ and $0 \leq j, l \leq W - w_k$, we add the clause:

$$\neg x_k \vee \neg (c_{ij}^k \wedge c_{ml}^k) \tag{6.9}$$

This second set of clauses ensures that a piece k is located in at most one cell. Otherwise, x_k should be false.

The second block of hard clauses ensures that pieces do not overlap with the location of the obstacles. For every piece k and every cell (i, j) filled with an obstacle, we add the clauses:

$$\neg c_{lm}^k \tag{6.10}$$

where $i \leq l \leq i + h_k - 1$ and $j - w_k + 1 \leq m \leq j$.

The third block of hard clauses ensures that any two pieces s and k do not overlap: for every allowed position (i, j) ($h_s \leq i \leq H$ and $0 \leq j \leq W - w_s$) of piece s, we add the following clauses:

$$\neg (c_{ij}^s \wedge c_{lm}^k) \tag{6.11}$$

for each allowed position (l, m) ($h_k \leq l \leq H$ and $0 \leq m \leq W - w_k$) of piece k, and where $i - h_s + 1 \leq l \leq i + h_k - 1$ and $j - w_k + 1 \leq m \leq j + w_s - 1$.

Finally, as in encoding BP1, there is a block of soft clauses. As we want to maximize the number of pieces in every bin taking into account the size of the pieces, we add the area of the piece as weight. So, the block of soft clauses is formed by the following weighted unit clauses:

$$(x_1, h_1 \times w_1), \ldots, (x_n, h_n \times w_n) \tag{6.12}$$

In encoding BP2, the set of clauses 6.9 encode the AMO constraint, since they ensure that if a piece k is in the bin, it is located in at most one cell. We also distinguish between two variants, the first one, called BP2-AMO, is encoding BP2 with the AMO clauses, and the second one, called BP2-NAMO, is encoding BP2 without the AMO clauses.

6.2 Experimental Investigation

We report on the experimental investigation conducted to analyse and compare the scaling behaviour of our generators with the Weighted Partial MaxSAT solvers WMaxSatz [LMP07b], MSUnCore [MSP08] and SAT4JMaxSAT [LP10]. All the experiments were performed on a Linux Cluster where the nodes have a 2GHz AMD Opteron processor with 1Gb of RAM.

6.2.1 Max1+pSAT

We present experimental results for the average complexity of solving random instances of Max1+pSAT. The number of clauses (C) of the instances has been set to depend on the number of variables (V) with the formula $C = 2 \cdot V \ln(V)$. Such a formula has been chosen to coincide with the size of the instances that we use for the MaxCut problem in Subsection 6.2.3, in which the number of edges has been set to the minimum possible that allow a random graph to be connected. The aim of having instances with the same, or similar, number of variables and number of clauses from

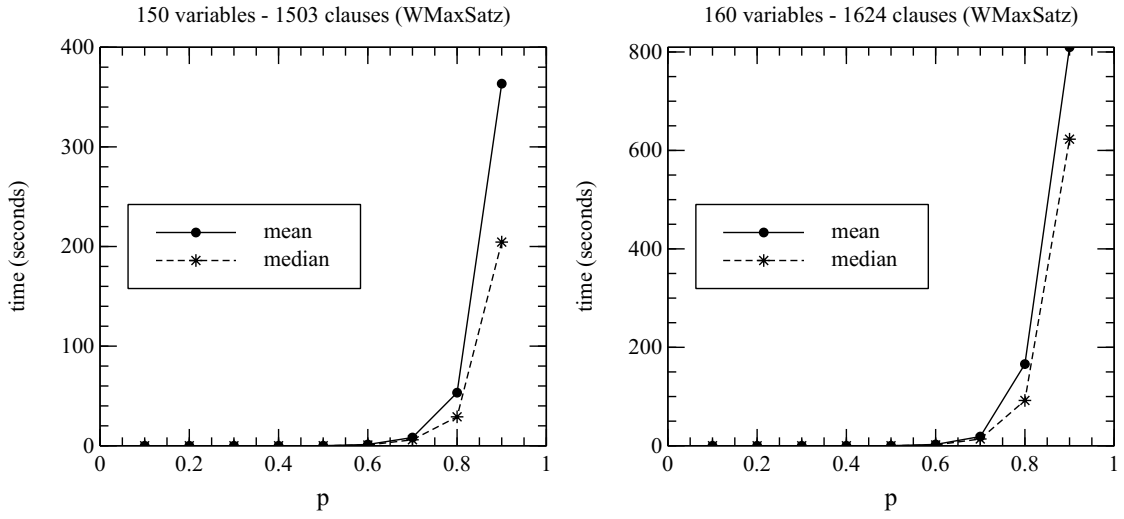

Figure 6.3: Random Max1+pSat solved with WMaxSatz.

different problem generators is to compare the complexity of instances of similar size, in such a way that only the structure of the problem can make a difference in the computational difficulty of solving an instance.

Figure 6.3 shows the mean and median time for solving random Max1+pSAT instances with 150 and 160 variables with WMaxSatz. We have generated sets of 100 instances for each different value of p, starting with $p = 0.1$ and ending with $p = 0.9$ and increasing it in steps of 0.1. Along the horizontal axis is the value of p in the instances, and the vertical axis shows the mean, or median, time needed to solve each set of 100 instances. We observe a clear exponential increase in time as the percentage of binary clauses increases. Interestingly, it seems that around the value $p = 0.7$ is where there is an abrupt change in the complexity of solving the instances.

A similar experiment was performed in [SW02], but the authors did not observe any exponential increase. We believe that this was due to the fact they used a MaxSAT solver that did not incorporate a good quality lower bound.

Figure 6.4 shows the results of solving random Max1+pSAT instances with MSUnCore. In this case, the results are for a number of variables equal to 25 and 28, because

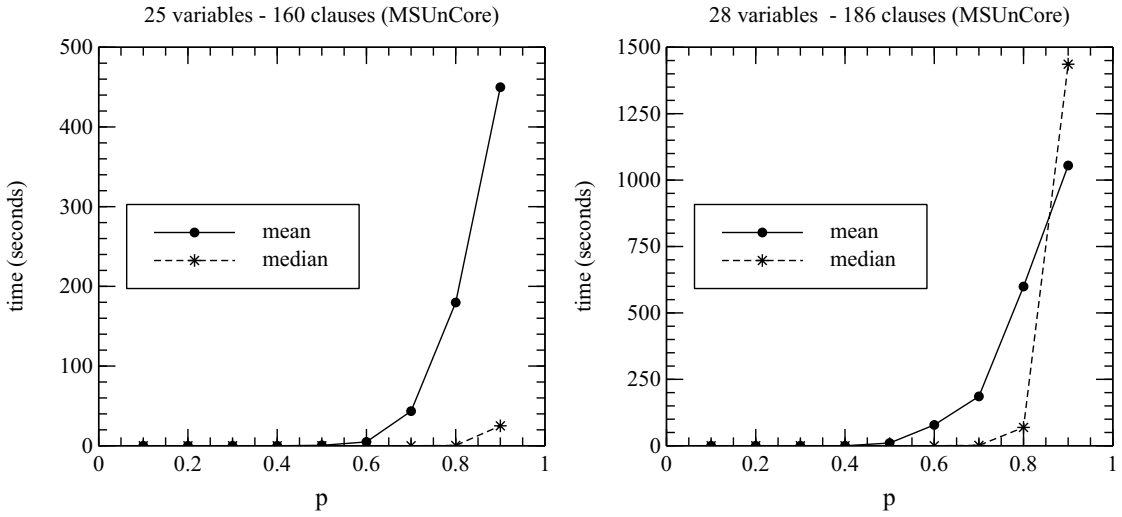

Figure 6.4: Random Max1+pSat solved with MSUnCore.

MSUnCore has a worse scaling cost than WMaxSatz, so the range of sizes we can solve is smaller. We observe an analogous behaviour, that is, an abrupt change in complexity around $p = 0.7$.

Finally, Figure 6.5 shows the results of solving random Max1+pSAT instances with SAT4JMaxSAT. In this case, the results are for a number of variables equal to 28 and 29. We observe a behaviour similar to the behaviour of MSUnCore: there is an abrupt change around $p = 0.7$, and the scaling cost is worse than the scaling cost of WMaxSatz.

We have also performed more experiments increasing the number of variables to study the scaling behaviour of the solvers. Table 6.1 shows the scaling of the mean and median cost as the number of variables increases for WMaxSatz. We show the results for both the hardest point ($p = 0.9$) and for the point $p = 0.7$, which is the point where an abrupt change in the complexity occurs for all the solvers. In the table, a dash in the column for the mean indicates that more than 20% of the instances were not solved within the cutoff time we used (1800 seconds), and in the column for the median the value $> 1800s$ indicates than more than 50% of the instances were not solved within

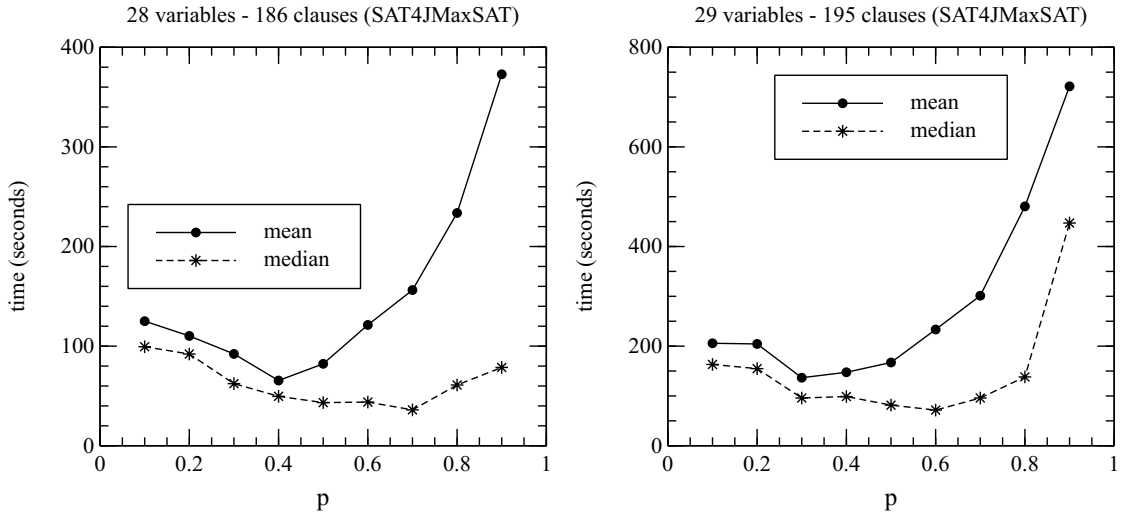

Figure 6.5: Random Max1+pSat solved with SAT4JMaxSAT.

the cutoff time. We observe a clear exponential increase in the mean and median time as V increases for both values of p. Table 6.2 shows analogous results for MSUnCore and SAT4JMaxSAT for the same values of p, but with a different range of values for V. We also observe the exponential increase in mean and median time for both values of p. Observe that the sizes that we can solve with WMaxSatz with respect to the ones we can solve with MSUnCore and SAT4JMaxSAT are very different. Our results are consistent with previous results obtained in MaxSAT evaluations where BnB solvers outperform SAT-based solvers on pure random instances [ALMP11b].

6.2.2 Partial Max2SAT

We present experimental results for the average complexity of solving random instances of Partial Max2SAT. As in the previous problem, the number of clauses (C) of the instances has been set to depend on the number of variables (V) using the formula $C = 2 \cdot V \ln(V)$.

Figure 6.6 shows the mean and median time for solving random Partial Max2SAT instances with 130 and 140 variables with WMaxSatz. As we observed that the change

	WmaxSatz			
	$p = 0.9$		$p = 0.7$	
	mean	median	mean	median
V=140, C=1383	99	53.86	3.34	2.48
V=150, C=1503	363	204	8.27	6.00
V=160, C=1624	809	622	18.70	13.78
V=170, C=1746	-	>1800	65.21	33.5

Table 6.1: Performance of WMaxSatz on hard instances for Max1+pSat. Time in seconds.

	MSUnCore				SAT4JMaxSAT			
	$p = 0.9$		$p = 0.7$		$p = 0.9$		$p = 0.7$	
	mean	median	mean	median	mean	median	mean	median
V=20, C=119	37	0.06	0.06	0.01	2.82	2.21	1.84	1.72
V=25, C=160	450	25	43.43	0.03	22.88	18.63	15.44	7.84
V=28, C=186	1054	1436	185	1.45	372	99.41	156	32.08
V=30, C=204	-	>1800	442	19.33	-	>1800	630	218

Table 6.2: Performance of MSUnCore and SAT4JMaxSAT on hard instances for Max1+pSAT. Time in seconds.

in complexity is driven by the ratio of number of hard clauses to number of variables, we generated a set of 100 instances for each different ratio, starting with ratio 0.0 and ending with ratio 1.5. Along the horizontal axis is the ratio of number of hard clauses to number of variables for each set of instances. The plots labelled as *mean* and *median* show the mean and median time for each set of instances, with the scale shown in the left vertical axis, and the plot labelled as *% of SAT instances* shows the percentage of instances that are feasible (that have a solution that satisfies all the hard clauses) for each set of instances, with the scale shown in the right vertical axis.

We observe that the higher the ratio of hard clauses to variables is, the lower the average complexity of solving the instances. Observe that the decrease in complexity is very abrupt up to approximetely the ratio 0.5, and then almost does not change. It seems that, at such ratio, WMaxSatz is able to quickly prune many branches with partial assignments inconsistent with the hard clauses. Observe that the plot for percentage of satisfiable instances indicates that at the ratio 0.5 we are very near to the beginning of the region where unsatisfiable instances begin to appear. So, it is consistent with the idea of having an increase in the number of partial assignments that are inconsistent with the hard clauses.

Figure 6.7 shows the results of solving random Partial Max2SAT instances with SAT4JMaxSAT. In this case, the results are for a number of variables equal to 26 and 28, because SAT4JMaxSAT, as in the previous problem, has a worse scaling cost than WMaxSatz, so the range of sizes we can solve is smaller. We observe an analogous behaviour: the complexity decreases as the ratio of hard clauses to variables increases, and the decrease is very abrupt up to to approximetely the ratio 0.5.

Finally, Figure 6.8 shows the results of solving random Partial Max2SAT instances with MSUnCore. In this case, the results are for a number of variables equal to 21 and 22. Observe that, for this solver, the decrease in the complexity seems to be linear with respect to the ratio of hard clauses to variables, in constrast to the previous exponential decrease in the complexity, at least until we arrive to a higher ratio than

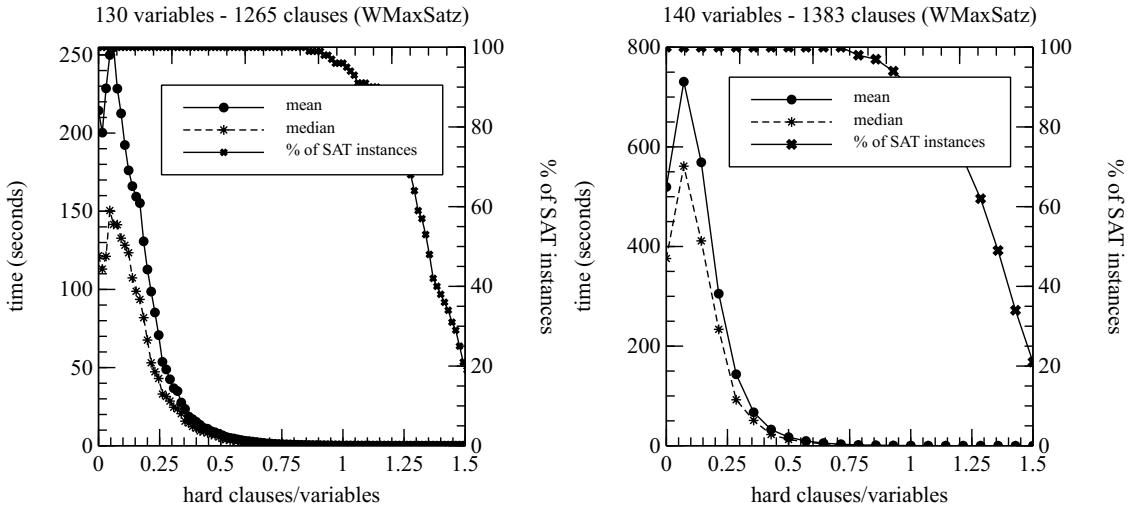

Figure 6.6: Partial Max2SAT solved with WMaxSatz.

before (1.5 in this case). It seems that MSUnCore is not able to take advantage of the sudden change in the characteristics of the instances that we observe with WMaxSatz around the ratio 0.5.

As with the previous problem, we have also studied the scaling behaviour of the solvers. Table 6.3 shows the scaling of the mean and median cost as the number of variables increases for WMaxSatz. We show the results for the hardest point, which is when almost all the clauses are soft. As before, we observe a clear exponential increase in the mean and median time as V increases. Table 6.4 shows analogous results for MSUnCore and SAT4JMaxSAT but for different ranges of values for V. We also observe the exponential increase in mean and median time as V increases. Again, we observe that the sizes that we can solve with WMaxSatz with respect to the ones we can solve with MSUnCore and SAT4JMaxSAT are very different.

WMaxSatz		
	mean	median
V=120, C=1148	55	37.29
V=130, C=1265	254	150
V=140, C=1383	730	561
V=160, C=1624	-	>1800

Table 6.3: Performance of WMaxSatz on the hardest instances for Partial Max2SAT. Time in seconds.

MSUnCore		
	mean	median
V=20, C=119	250	3.34
V=21, C=127	366	16.2
V=22, C=136	654	135
V=23, C=144	-	891

SAT4JMaxSAT		
	mean	median
V=20, C=119	3.26	2.27
V=23, C=144	6.25	4.61
V=26, C=169	124	20.16
V=30, C=204	-	>1800

Table 6.4: Performance of MSUnCore and SAT4JMaxSAT on the hardest instances for Partial Max2SAT. Time in seconds.

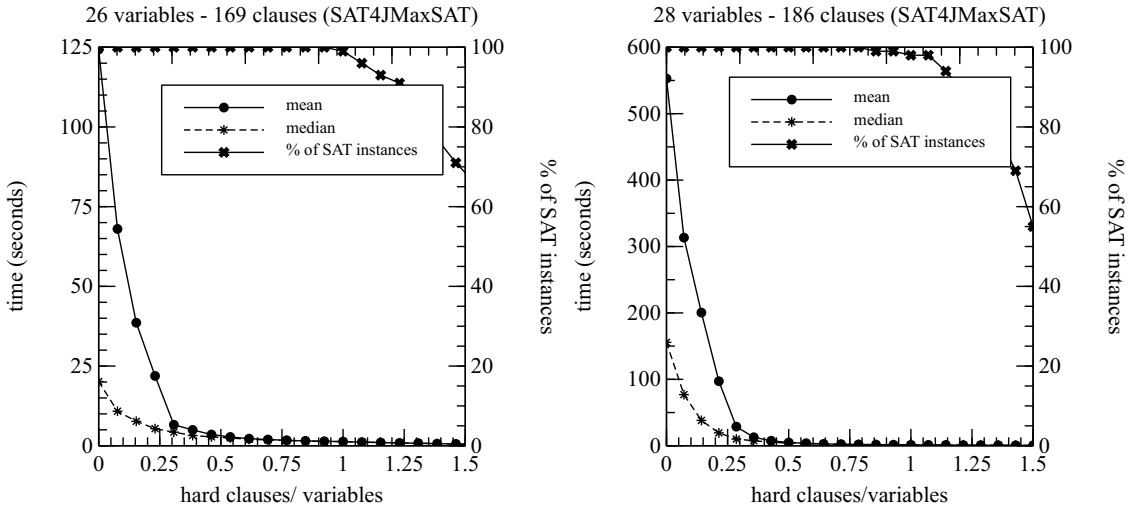

Figure 6.7: Partial Max2SAT solved with SAT4JMaxSAT.

6.2.3 MaxCut

We present experimental results for the average complexity of solving instances of MaxCut with the generator of random graphs we have described in Subsection 6.1.3. Recall that the generator creates a graph with V vertices, V always even, and m edges by first building a bipartite graph with $(1-p)\cdot m$ edges and high expansion, with the algorithm presented in [ABFM08], and then adding a random set of $p \cdot m$ edges which may destroy the bipartition of the graph. The intention of the parameter p is the following: when $p = 0$, the graph is bipartite so the maximum cut should be easy to find once we discover that the graph is bipartite. As p increases, the bipartition can disappear so the maximum cut may not be the same cut as the cut of the initial bipartite graph. The idea is that if the original bipartite graph has high expansion, it may still contain many sub-optimal cuts that are very close to the maximum cut, so this may create difficulties when trying to find, and certify, the maximum cut.

In the generated instances, the number of edges m has been set to $V\ln(V)$, so the number of clauses (C) is $2 \cdot V\ln(V)$. This number of edges is just to the right of the phase transition for the connectedness of a random graph [Bol01]. In this way,

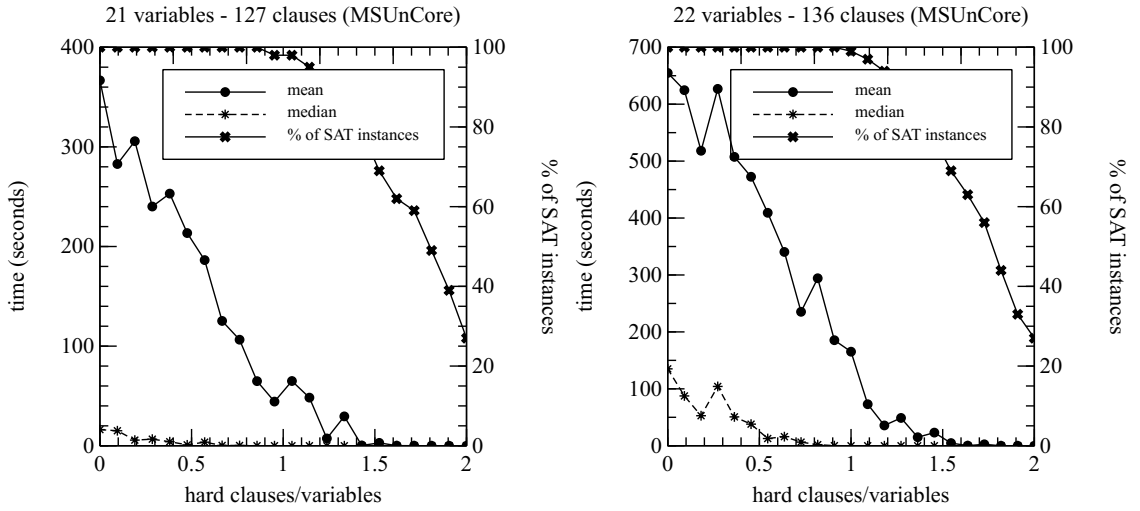

Figure 6.8: Partial Max2SAT solved with MSUnCore.

we ensure that at least the graph will be connected when all the edges are randomly generated ($p = 1$), and the problem will be not decomposable in smaller subproblems.

Figure 6.9 shows the mean and median time for solving random MaxCut instances with 130 and 136 vertices with WMaxSatz. We have generated sets of 100 instances for each different value of p, starting with 0.1 and ending with 1, and increasing it in steps of 0.1. Along the horizontal axis is the value of p in the instances, and the vertical axis shows the mean, or median, time needed to solve each set of 100 instances. The results show clearly that, when the graphs are almost bipartite, WMaxSatz finds easily the maximum cut, but as p increases the complexity increases, up to a point, around $p = 0.5$, where the complexity seems to decrease but not very abruptly. We believe that in the point of maximum complexity the graphs are not bipartite, but still many suboptimal cuts are found in the original bipartite subgraph, due to its high expansion. As p increases further, these suboptimal cuts are probably lost, so the complexity of finding and certifying the maximum cut decreases.

Figure 6.10 shows the results of solving random MaxCut instances with 24 and 26 vertices with SAT4JMaxSAT. We observe an analogous behaviour: the complexity increases up to approximetely the point $p = 0.5$, and then decreases.

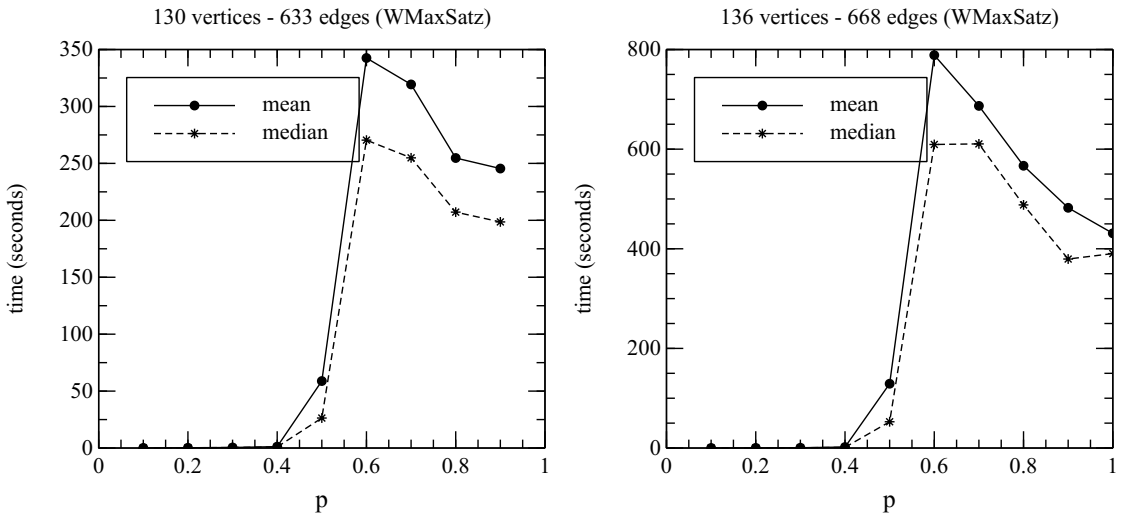

Figure 6.9: MaxCut solved with WMaxSatz.

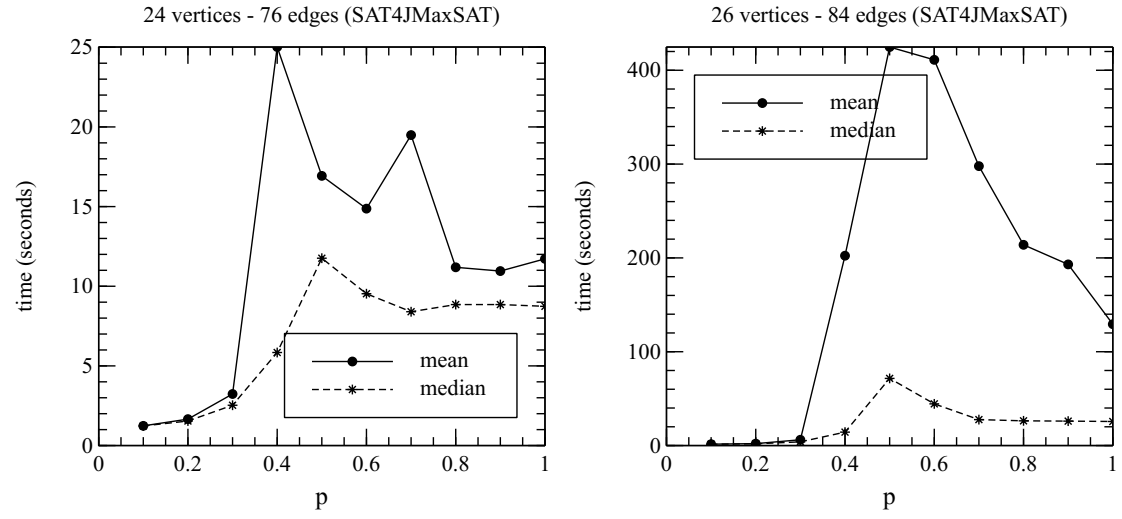

Figure 6.10: MaxCut solved with SAT4JMaxSAT.

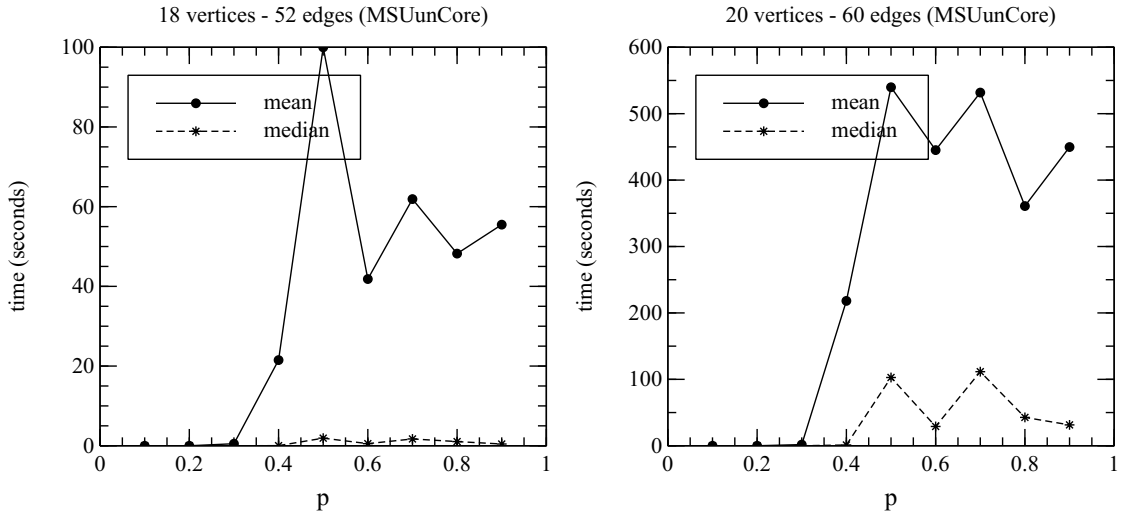

Figure 6.11: MaxCut solved with MSUnCore.

Figure 6.11 shows the results of solving random MaxCut instances with 18 and 20 vertices with MSUnCore, that are smaller sizes than for SAT4JMaxSAT because its performance is much worse. In this case, even if the complexity also increases up to approximetely the point $p = 0.5$, then the decrease is not so sharp. We believe this is mainly due to the small size of the instances we have been able to solve with MSUnCore with our cutoff time of 1800 seconds per instance.

As with the previous problem, we have also studied the scaling behaviour of the solvers. Table 6.5 shows the scaling of the mean and median cost as the number of vertices increases for WMaxSatz. We show the results for the hardest point, that is always around the point $p = 0.5$. As before, we observe a clear exponential increase in the mean and median time as V increases. Table 6.6 shows analogous results for MSUnCore and SAT4JMaxSAT but for different ranges of values for V. We also observe the exponential increase in mean and median time as V increases. Again, we observe that WMaxSatz scales up better than MSUnCore and SAT4JMaxSAT when the size grows, but in this case the scaling behaviour of MSUnCore is significantly worse than the scaling behaviour of SAT4JMaxSAT.

	WMaxSatz	
	mean	median
V=120, m=574	100	76.9
V=130, m=633	343	270
V=136, m=668	788	610
V=140, m=692	1013	951

Table 6.5: Performance of WMaxSatz for the hardest instances of MaxCut. Time in seconds.

	MSUnCore			SAT4JMaxSAT	
	mean	median		mean	median
V=16, m=44	8	0.05	V=18, m=52	1.67	1.58
V=18, m=52	100	1.91	V=20, m=60	2.54	2.4
V=20, m=60	540	111	V=24, m=76	25.01	11.76
V=22, m=136	-	> 1800	V=26, m=84	424	71.56

Table 6.6: Performance of MSUnCore solver and SAT4JMaxSAT solver for the hardest instances of MaxCut. Time in seconds.

6.2.4 Bin packing

We present experimental results for the average complexity of solving instances of bin packing with the generator of random instances described in Subsection 6.1.4. In order to bound the number of parameters to set and control the complexity of the instances, the pieces are always generated with a minimum height (width) 1 and maximum height (width) 2. This way, the pieces are very similar in size, so we try to avoid that the solver takes profit of big differences between piece sizes. Then, given a bin size $H \times W = A$, the piece set is randomly generated bounding the sum of the n piece areas to be in the range $[0.95 \cdot A/2, 1.05 \cdot A/2]$. With this aim, the number of pieces (n) is always chosen in our experiments such that the expected value for the sum of the areas ($n \cdot 1.5 \cdot 1.5$) to be as close as possible to the value $A/2$. We present results for two different bin sizes: 7×7 and 8×8 in our experiments, given that 6×6 instances are too easy and 9×9 instances are too hard for our solvers.

First, we present the results obtained with WMaxSatz for 7×7 bin size and number of pieces (n) 10. Figure 6.12 shows the mean time needed to solve test-sets of 100 instances with the four variants of the encoding we have considered: BP1-AMO, BP1-NAMO, BP2-AMO and BP2-NAMO. The results for the median time are qualitatively similar to the mean, so in order to not present an excesive amount of plots we only discuss results for the mean in this subsection. Each test-set considers a different occupancy percentage of obstacles with respect to the total bin size area. This occupancy percentage is the value shown in the x-axis of the plots, and the y-value is the mean time needed to solve each set of 100 instances. For the encodings of BP1 we observe that, as we increase the percentage of obstacles up to approximetely the percentage of 70%, the mean time increases and then it starts to decrease slightly. We believe that this behaviour is due to the fact that as the expected value for the sum of the pieces areas is half the bin size area, when the percentage of obstacles is far below half the bin size area (50% point) optimal solutions will tend to have almost all the pieces and there will be many optimal solutions (different ways of placing the pieces in the bin). Then, as the percentage of obstacles approaches the

50% point, the number of pieces that can be placed will decrease, and it will increase the difficulty of discovering the pieces of an optimal solution. When the percentage of obstacles is far above the 50% point, optimal solutions will tend to have very few pieces. To corroborate this possible relation between the hardness and the set of possible solutions, we also show in the figure the mean of the optimal value normalized by its maximum value, that is normalized by the sum of the piece areas of the instance. We observe clearly that, for a low percentage of obstacles, almost all the pieces can be located, but as more obstacles are placed, the number of pieces quickly descends coinciding with the increase in the hardness of solving the instances. When we arrive at the peak of hardness, we observe that the number of pieces in optimal solutions is around half the total (considering that all the pieces have almost the same size as in our experiments).

For the encodings of BP2, it seems that the behaviour is very different. Overall, the complexity decreases with respect to BP1, and it seems to have a no clear intermediate peak of hardness, but as we will see next the peak appears when the size increases. We think that the performance of encoding BP2 is better than the performance of BP1 because it helps prune many inconsistent assignments with unit propagation. In encoding BP1, a piece k is located at position (i, j) by setting two variables (r_i^k and c_j^k) to true. Consider a piece k such that no available position (i, j) in the bin can fit the piece. In encoding BP2 this would generate a unit clause $(\neg x_k)$ because all the variables encoding possible locations for k would be propagated to be false thanks to the non-overlaping clauses $(\neg c_{ml}^s \vee \neg c_{ij}^k)$ it has with other pieces s. But in encoding BP1 the same situation will not necessarily generate the unit clause $(\neg x_k)$, because if for piece k there is at least one row variable r_i^k available and at least one column variable c_j^k available, even if they cannot be both true, then this will not generate immediatly the unit clause $(\neg x_k)$.

Next we present results for MSUnCore and SAT4JMaxSAT for the same problem instances. Figure 6.13 shows the results. Qualitatively, we observe a behaviour analogous to WMaxSatz, although the existence of a peak in the mean time is more

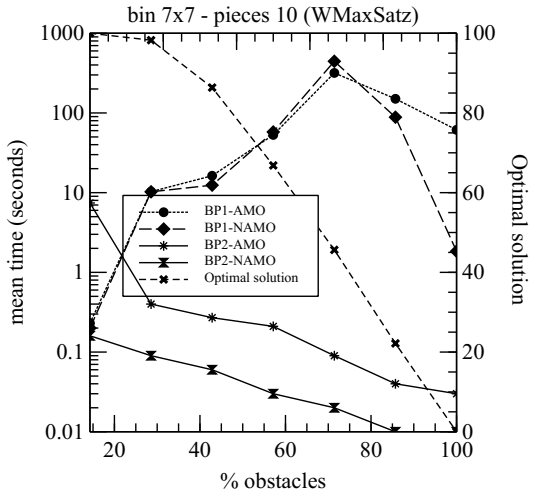

Figure 6.12: Bin packing problem, 7x7, 10 pieces. Results for WmaxSatz.

evident in SAT4JMaxSAT than in MSUnCore. Nevertheless, as we observe in the next results, the peak is more clearly observed as the bin size increases. At the same time, the performance difference between the encodings BP1 and BP2 is not so large, so it seems that for these SAT-based MaxSAT solvers the advantatge in propagation that BP2 gives is not as useful as with BnB solvers.

Finally, we present results for 8×8 bin size and number of pieces (n) 15 solved with WMaxSatz, MSUnCore and SAT4JMaxSAT. Figure 6.14 shows the results for these solvers when using the best encoding for each one. We observe that encoding BP2 is in any case the best one, and that now we observe more clearly the peak of hardness for an intermediate value of the percentage of obstacles. Observe that the SAT-based algorithms perform better than WMaxSatz. These results are consistent with previous results about the performance of SAT-based and BnB solvers for crafted problems obtained in the MaxSAT Evaluation [ALMP11b].

6.3 Summary

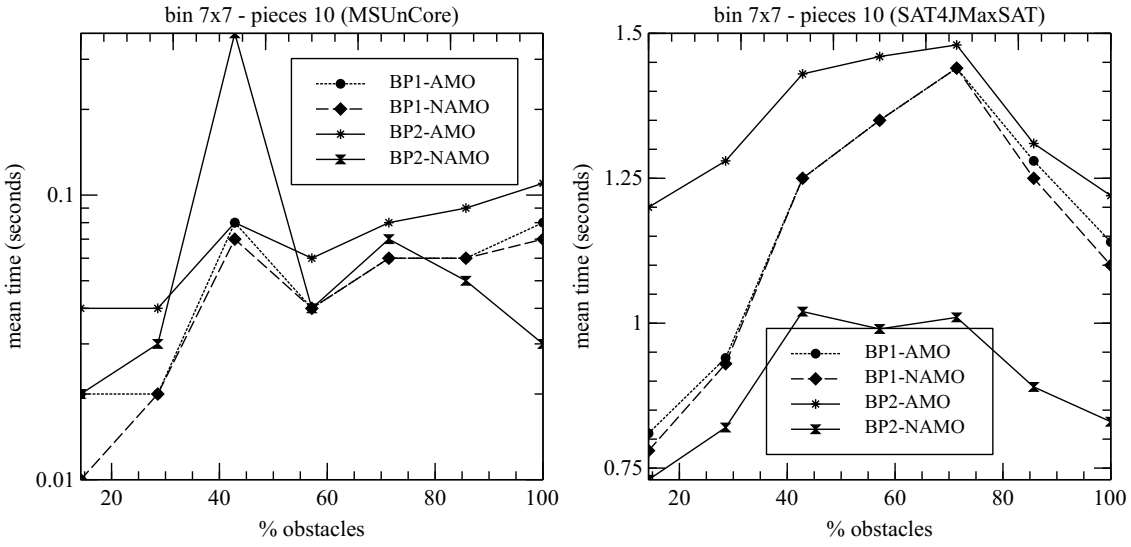

Figure 6.13: Bin packing problem, 7x7, 10 pieces. Results for MSUnCore (left) and for SAT4JMaxSAT (right).

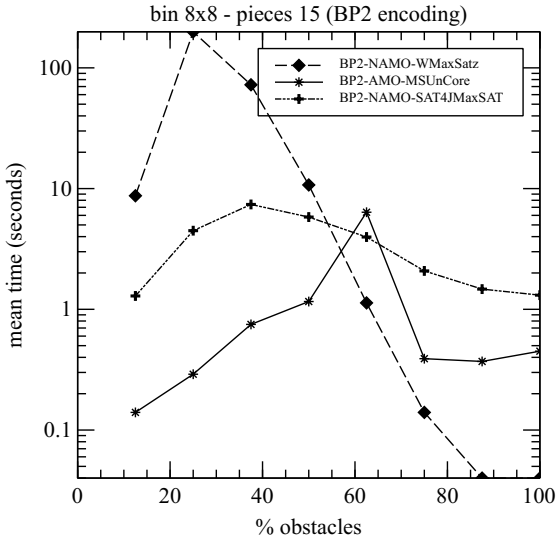

Figure 6.14: Bin packing problem, 8x8, 15 pieces. Results for WMaxSatz, MSUnCore and SAT4JMaxSAT with their best encodings.

In this chapter we have presented two generators of instances for MaxSAT and two for Weighted Partial MaxSAT. Our goal was to get new benchmark generators that allow to produce many instances of increasing difficulty by easily tunning some parameters, and without needing to scale to excessivily large sizes. We have performed an experimental evaluation of the hardness of the instances obtained with such generators.

For MaxSAT, the results obtained with the Max1+pSAT generator show an abrupt increase in the difficulty of the instances as the parameter p reaches the value 0.7. This happens with all the state-of-the-art MaxSAT solvers we have tried, so that we do not need to go beyond instances with more than 160 variables, when using BnB solvers, and with more than 30 variables, when using SAT-based solvers, to have mean solving times greater than 600 seconds. The results obtained with the MaxCut generator show an abrupt increase when its parameter p reaches the value 0.6, but beyond that value it starts to softly decrease. Analogously to the Max1+pSAT generator, with BnB solvers with instances with around 140 variables the mean solving times are around 800 seconds, and with SAT-based solvers with instances with around 30 variables the mean solving times are around 500 seconds. It is worth noticing that the effect of the parameter p seems to be slightly different in both problem generators. For Max1+pSAT, the higher the value of p, the higher the percentage of binary clauses, that are the ones that turn the problem NP-hard. By contrast, for our MaxCut generator, increasing the value of p increases the randomness of the graph, in such a way that some level of randomness increases the difficulty of the instances, compared to the more structured ones with $p = 0$ (the high expanding bipartite graphs). But as the random part of the graph starts to overtake the structured part, the mean difficulty decreases. This is an interesting difference between both generators, that it would be worth to investigate in more detail in the future.

For Partial MaxSAT, the results obtained with the Partial Max2SAT generator show that the average complexity of solving instances depends on the ratio of hard clauses to variables in the instance. In all the solvers we have tried, as this ratio

increases, the average complexity of solving instances decreases. Both WMaxSatz and SAT4JMaxSAT present an exponential complexity decrease. This decrease changes abruptly approximetely up to the ratio 0.5. By contrast, in MSUnCore, the complexity decrease is not so abrupt and it seems to be linear. Analogously to the previous MaxSAT generators, with WMaxSatz with instances around 140 variables the mean solving times are around 800 seconds, meanwhile with SAT-based solvers, we only are able to solve instances having around 28 variables for the same mean solving times.

For the bin packing generator, the results obtained show that the complexity of solving instances increases as we increase the percentage of obstacles up to approximetely 70%, and then it starts to decrease slightly. Overall, this fact is observed for both encodings, BP1 and BP2, although this behaviour is more clearly observed as the size increases. With respect to the relative performance of the different encodings, BP2 is the best for WMaxSatz, and as the size increases, it is also the best encoding for the three solvers.

Finally, our results provide empirical evidence that SAT-based algorithms perform better than WMaxSatz. This is consistent with previous results about the performance of SAT-based and BnB solvers for crafted problems obtained in the MaxSAT Evaluation. Our bin packing generator is a good option for a crafted problem generator of Weighted Partial MaxSAT instances, in case we need to test sets of instances with always feasible solutions, and where the hardness of the instances comes from the optimization part of the problem. Observe that in contrast to the other three MaxSAT generators, with this crafted problem generator the size of the instances we have solved with the solvers is considerably larger. For example, around 960 variables for the 8x8 bin size instances with encoding BP2 and around 240 variables with encoding BP1. So, if one wants test sets of hard but small instances the best option seems to be between the three first generators.

Chapter 7

Conclusions

In this thesis, we have defined original encodings from CSP into SAT, and have studied, for the first time, encodings from MaxCSP into MaxSAT. We have proved the correctness of the new encodings, analyzed their properties, and performed an empirical comparison that provides evidence of the good performing behaviour of the defined encodings. Moreover, we have designed, implemented and evaluated four generators of MaxSAT instances, which are able to produce a large amount of instances of adjustable computational difficulty.

The main contributions of our research work can be summarized as follows:

- The definition of two new encodings from CSP into SAT: the minimal support encoding, which reduces the size of the support encoding, and the interval-based support encoding, which is the first polynomial size support encoding containing only regular literals.

- The definition of encodings from MaxCSP into Partial MaxSAT that extend the existing direct and support encodings from CSP into SAT, as well as the analysis of their properties.

- An empirical evaluation of the existing and new encodings from CSP into SAT, and from MaxCSP into Partial MaxSAT. The obtained results provide empirical evidence of the good performance profile of the new encodings on SAT solvers such as MiniSat and PrecoSAT that incorporate conflict clause learning, as

well as of the good performance profile when they are used to encode MaxCSP into Partial MaxSAT, and are solved with MaxSAT solvers such as MSUnCore, SAT4J-Maxsat and WMaxSatz.

- The design and implementation of MaxSAT instance generators of adjustable hardness. The generators encode into MaxSAT the following combinatorial optimization problems: Max1+pSAT, Partial Max2SAT, MaxCut, and rectangular bin packing. The empirical evaluation of the proposed generators shows that they produce challenging and suitable benchmarks for testing MaxSAT solvers.

Finally, we would like to point out some future research perspectives:

- In the weighted constraint programming community, different levels of soft local consistency have been defined. It could be interesting to analyze the level of soft local consistency that our encodings from MaxCSP into MaxSAT achieve when they are solved with the modern branch-and-bound MaxSAT solvers that apply MaxSAT resolution refinements at each node of the search space.

- The Minimum Satisfiability problem (MinSAT) has been recently used to solve combinatorial optimization problems, and has shown to be superior to MaxSAT-based problem solving on certain classes of problems [LMQZ10, LZMS12, LZMS11]. We propose to define well-suited encodings from MaxCSP into MinSAT, as well as empirically compare their performance with the encodings of this thesis.

- To promote and facilitate the research on MaxSAT, we plan to create a publicly available repository of MaxSAT benchmarks including a description of the problems and their encodings, the optimal solutions, references to the papers that have used the instances, and a collection of generators of random instances.

Bibliography

[ABFM08] Carlos Ansótegui, Ramón Béjar, César Fernàndez, and Carles Mateu. From high girth graphs to hard instances. In *Principles and Practice of Constraint Programming, CP 2008*, pages 298–312, 2008.

[ABL09] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT-2009, Swansea, UK*, pages 427–440. Springer LNCS 5584, 2009.

[ABL10] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial MaxSAT. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence, AAAI-2010, Atlanta, USA*, pages 3–8, 2010.

[ACLM09a] Josep Argelich, Alba Cabiscol, Inês Lynce, and Felip Manyà. Regular encodings from Max-CSP into Partial Max-SAT. In *Proceedings, 39th International Symposium on Multiple-Valued Logics (ISMVL), Okinawa, Japan*, pages 196–202. IEEE CS Press, 2009.

[ACLM09b] Josep Argelich, Alba Cabiscol, Inês Lynce, and Felip Manyà. Sequential encodings from Max-CSP into Partial Max-SAT. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), Swansea, UK*, pages 161–166. Springer LNCS 5584, 2009.

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

[ALMP11a] Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. Analyzing the instances of the MaxSAT evaluation. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing, SAT-2011, Ann Arbor, MI/USA*, pages 360–361. Springer LNCS 6695, 2011.

[ALMP11b] Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. Analyzing the instances of the maxsat evaluation. In *Theory and Applications of Satisfiability Testing - (SAT 2011)*, pages 360–361, 2011.

[ALMP11c] Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. Experimenting with the instances of the MaxSAT evaluation. In *Proceedings of the 14th International Conference of the Catalan Association for Artificial Intelligence, CCIA-2011, Lleida, Spain*, pages 31–40. IOS Press, 2011.

[AM04] Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables into problems with Boolean variables. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT), Vancouver, Canada*, pages 1–15. Springer LNCS 3542, 2004.

[AMP03] Teresa Alsinet, Felip Manyà, and Jordi Planes. Improved branch and bound algorithms for Max-SAT. In *Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing*, 2003.

[AMP04] Teresa Alsinet, Felip Manyà, and Jordi Planes. A Max-SAT solver with lazy data structures. In *Proceedings of the 9th Ibero-American Confer-*

ence on Artificial Intelligence, IBERAMIA 2004, Puebla, México, pages 334–342. Springer LNCS 3315, 2004.

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

[AMP08] Teresa Alsinet, Felip Manyà, and Jordi Planes. An efficient solver for Weighted Max-SAT. *Journal of Global Optimization*, 41:61–73, 2008.

[Arg08] Josep Argelich. *Max-SAT Formalisms with Hard and Soft Constraints*. PhD thesis, Department of Computer Science and Industrial Engineering. Universitat de Lleida, 2008.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solver. In *Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, pages 399–404, 2009.

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

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *9th International Conference on Principles and Practice of Constraint Programming, CP-2003, Kinsale, Ireland*, pages 108–112. Springer LNCS 2833, 2003.

[BB09] Hans Kleine Büning and Uwe Bubeck. Theory of quantified boolean formulas. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 735–760. IOS Press, 2009.

[Ber] Daniel Le Berre. SAT4J, a satisfiability library for java. http://www.sat4j.org/.

[BGS99] Laure Brisoux, Eric Gregoire, and Lakhdar Sais. Improving backtrack search for sat by means of redundancy. In *Foundations of Intelligent Systems, 11th International Symposium, (ISMIS-99)*, pages 301–309, 1999.

[BHM00] Bernhard Beckert, Reiner Hähnle, and Felip Manyà. The SAT problem of signed CNF formulas. In David Basin, Marcello D'Agostino, Dov Gabbay, Seán Matthews, and Luca Viganò, editors, *Labelled Deduction*, volume 17 of *Applied Logic Series*, pages 61–82. Kluwer, Dordrecht, 2000.

[BHM01] Ramón Béjar, Reiner Hähnle, and Felip Manyà. A modular reduction of regular logic to classical logic. In *Proceedings, 31st International Symposium on Multiple-Valued Logics (ISMVL), Warsaw, Poland*, pages 221–226. IEEE CS Press, Los Alamitos, 2001.

[BHW03] Christian Bessière, Emmanuel Hebrard, and Toby Walsh. Local consistencies in SAT. In *Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing (SAT), Santa Margarita Ligure, Italy*, pages 299–314. Springer LNCS 2919, 2003.

[Bie10a] Armin Biere. Plingeling version 276. See http://fmv.jku.at/lingeling/, 2010.

[Bie10b] Armin Biere. PrecoSAT version 236. See http://fmv.jku.at/precosat/, 2010.

[BK02] Armin Biere and Wolfgang Kunz. Sat and atpg: Boolean engines for formal hardware verification. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, ICCAD-2002, San Jose, California, USA*, pages 782–785. ACM, 2002.

[BLM06] María Bonet, Jordi Levy, and Felip Manyà. A complete calculus for Max-SAT. In *Proceedings of the 9th International Conference on The-*

ory and Applications of Satisfiability Testing, SAT-2006, Seattle, USA, pages 240–251. Springer LNCS 4121, 2006.

[BLM07] María Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007.

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

[BMC+07] Ramón Béjar, Felip Manyà, Alba Cabiscol, César Fernández, and Carla P. Gomes. Regular-sat: A many-valued approach to solving combinatorial problems. *Discrete Applied Mathematics*, 115(12):1613–1626, 2007.

[Bol01] Béla Bollobás. *Random Graphs. Second Edition.* Number 73 in Cambridge studies in advanced mathematics. Cambridge University Press, 2001.

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

[BS97] Roberto J. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97, Providence/RI, USA*, pages 203–208. AAAI Press, 1997.

[BSST09] Clark Barret, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

[CA93] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI'93, Washington, D.C., USA*, pages 21–27. AAAI Press, 1993.

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

[CGLR96] James M. Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings ot the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.

[CS00] Philippe Chatalic and Laurent Simon. Zres: The old davis-putnam procedure meets zbdd. In David McAllester, editor, *17th International Conference on Automated Deduction (CADE'17)*, number 1831 in LNCS, pages 449–454, 2000.

[Cul] Joseph Culberson. Graph coloring page: Graph generator programs. See http://webdocs.cs.ualberta.ca/˜joe/Coloring/.

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

[DD01] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'01, Seattle/WA, USA*, pages 248–253, 2001.

[DDDL07] Sylvain Darras, Gilles Dequen, Laure Devendeville, and Chu Min Li. On inconsistent clause-subsets for Max-SAT solving. In *Proceedings of 13th*

International Conference on Principles and Practice of Constraint Programming, CP-2007, Providence, USA, pages 225–240. Springer LNCS 4741, 2007.

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

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

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, SAT-2003, Santa Margherita Ligure, Italy*, pages 502–518. Springer LNCS 2919, 2003.

[eSSMS99] Luís Guerra e Silva, Luis Miguel Silveira, and João P. Marques-Silva. Algorithms for solving boolean satisfiability in combinational circuits. In *Proceedings of Design, Automation and Test in Europe, DATE'99, Munich, Germany*, pages 526–530, 1999.

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

[FP01] Alan M. Frisch and Timothy J. Peugniez. Solving non-boolean satisfiability problems with stochastic local search. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'01, Seattle/WA, USA*, pages 282–290, 2001.

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

[Gav07] Marco Gavanelli. The log-support encoding of CSP into SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP-2007, Providence/RI, USA*, pages 815–822. Springer LNCS 4741, 2007.

[Gel02] A. Van Gelder. Generalizations of watched literals for backtracking search. In *Proceedings of the 7th International Symposium on Artificial Intelligence and Mathematics, Ft. Lauderdale, FL*, 2002.

[Gen02] Ian P. Gent. Arc consistency in SAT. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI), Lyon, France*, pages 121–125. IOS Press, 2002.

[GJ96] R. Génisson and P. Jégou. Davis and Putnam were already checking forward. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI), Budapest, Hungary*, pages 180–184. IOS Press, 1996.

[GMN09] Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. Reasoning with quantified boolean formulas. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 761–780. IOS Press, 2009.

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

[GW93] Ian Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for sat. In *Proceedings of National Conference on Artificial Intelligence (AAAI-93)*, pages 28–33, 1993.

[HDvMvZ04] Marijn Heule, Mark Dufour, Hans van Maaren, and Joris van Zwieten. March_eq: Implementing efficiency and additional reasoning into

a lookahead sat-solver. *Journal on Satisfiability, Boolean Modeling and Computation*, pages 25–30, 2004.

[HL06] Federico Heras and Javier Larrosa. New inference rules for efficient Max-SAT solving. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-2006, Boston/MA, USA*, pages 68–73, 2006.

[HLO07] Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: A new weighted Max-SAT solver. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing, SAT-2007, Lisbon, Portugal*, pages 41–55. Springer LNCS 4501, 2007.

[Hoo99] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence, AAAI'99*, pages 661–666. AAAI Press, 1999.

[HS04] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search. Foundations and Applications*. Morgan Kaufmann, 2004.

[HTH02] Frank Hutter, Dave Tompkins, and Holger Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Proceedings of CP-02*, volume 2470 of *LNCS*, pages 233–248. Springer, 2002.

[HV95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[Kas90] S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275–286, 1990.

[Kau06] Henry A. Kautz. Deconstructing planning as satisfiability. In *Proceedings of the 21st National Conference on Artificial Intelligence, AAAI-2006, Boston/MA, USA*, 2006.

[Kri85] Balakrishnan Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22(3):253–275, 1985.

[KS96] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'96, Portland/OR, USA*, pages 1194–1201, 1996.

[KSHK07] Daher Kaiss, Marcelo Skaba, Ziyad Hanna, and Zurab Khasidashvili. Industrial strength sat-based alignability algorithm for hardware equivalence verification. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design, FMCAD-2007, Austin/TX, USA*, pages 20–26. IEEE Computer Society, 2007.

[KSTW05] Philip Kilby, John K. Slaney, Sylvie Thibaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the Twentieth National Conference in Artificial Intelligence (AAAI-05)*, pages 1368–1373. AAAI Press, 2005.

[Kue10] Adrian Kuegel. Improved exact solver for the Weighted MAX-SAT problem. In *Proceedings of Workshop Pragmatics of SAT*, 2010.

[LA97a] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'97, Nagoya, Japan*, pages 366–371, 1997.

[LA97b] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the 3rd International Conference*

on Principles of Constraint Programming, CP'97, Linz, Austria, pages 341–355. Springer LNCS 1330, 1997.

[LH05] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, SAT-2005, St. Andrews, Scotland*, pages 158–172. Springer LNCS 3569, 2005.

[LHdG08] Javier Larrosa, Federico Heras, and Simon de Givry. A logical approach to efficient max-sat solving. *Artificial Intelligence*, 172(2–3):204–233, 2008.

[Li03] Chu Min Li. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130:251–276, 2003.

[LM09] Chu Min Li and F. Manyà. Max-sat, hard and soft constraints. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 613–631. IOS Press, 2009.

[LMMP10] Chu Min Li, Felip Manyà, Nouredine Ould Mohamedou, and Jordi Planes. Resolution-based lower bounds in MaxSAT. *Constraints*, 15(4):456–484, 2010.

[LMP05] Chu Min Li, Felip Manyà, and Jordi Planes. Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, CP-2005, Sitges, Spain*, pages 403–414. Springer LNCS 3709, 2005.

[LMP06] Chu Min Li, Felip Manyà, and Jordi Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for max-sat. In *Proceedings of the 21st National Conference on Artificial Intelligence, AAAI-2006, Boston/MA, USA*, pages 86–91, 2006.

[LMP07a] Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for max-sat. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

[LMP07b] Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

[LMQZ10] Chu Min Li, Felip Manyà, Zhe Quan, and Zhu Zhu. Exact MinSAT solving. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing, SAT-2010, Edinburgh, UK*, pages 363–368. Springer LNCS 6175, 2010.

[LMS01] Inês Lynce and Joao P. Marques-Silva. Integrating simplification techniques in sat algorithms. In *IEEE Symposium on Logic in Computer Science*, 2001. Short paper session.

[LMS02] Inês Lynce and Joao P. Marques-Silva. Efficient data structures for backtrack search SAT solvers. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing, SAT-2002, Cincinnati, USA*, pages 308–315, 2002.

[LMS05] Inês Lynce and João P. Marques-Silva. Efficient data structures for backtrack search sat solvers. *Annals of Mathematics and Artificial Intelligence*, 43(1):137–152, 2005.

[LMS06a] Inês Lynce and João Marques-Silva. Efficient haplotype inference with boolean satisfiability. In *Proceedings of the 21st National Conference on Artificial Intelligence, AAAI-2006, Boston/MA, USA*, 2006.

[LMS06b] Inês Lynce and João Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, SAT-2006, Seattle, USA*, pages 136–141. Springer LNCS 4121, 2006.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[LS07] Han Lin and Kaile Su. Exploiting inference rules to compute lower bounds for max-sat solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI-2007, Hyderabad, India*, pages 2334–2339, 2007.

[LSL08] Han Lin, Kaile Su, and Chu Min Li. Within-problem learning for efficient lower bound computation in Max-SAT solving. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence, AAAI-2008, Chicago/IL, USA*, pages 351–356, 2008.

[LWZ07] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for sat. In *Proceedings of 10th international conference on the Theory and Applications of Satisfiability Testing, SAT-2007, Lisbon, Portugal*, pages 121–133. Springer LNCS 5401, 2007.

[LZMS11] ChuMin Li, Zhu Zhu, Felip Manyà, and Laurent Simon. Minimum satisfiability and its applications. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI-2011, Barcelona, Spain*, pages 605–610, 2011.

[LZMS12] ChuMin Li, Zhu Zhu, Felip Manyà, and Laurent Simon. Optimizing with minimum satisfiability. *Artificial Intelligence*, 2012. http://dx.doi.org10.1016/j.artint.2012.05.004.

[Mat09] Carles Mateu. *CSP problems as algorithmic benchmarks: measures, methods and models*. PhD thesis, Department of Computer Science and Industrial Engineering. Universitat de Lleida, January 2009.

[Mat11a] Bryan Matsuo. EBGlucose version 1.0. See http://users.soe.ucsc.edu/ bmatsuo/ebglucose.html, 2011.

[Mat11b] Bryan Matsuo. EBMiniSAT. See http://users.soe.ucsc.edu/ bmatsuo/ebminisat.html, 2011.

[Mit05] David Mitchell. A sat solver primer. *European Association for Theoretical Computer Science (EATCS) Bulletin*, 85:112–133, 2005.

[MML10] Vasco M. Manquinho, Ruben Martins, and Inês Lynce. Improving unsatisfiability-based algorithms for Boolean optimization. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing, SAT-2010, Edinburgh, UK*, pages 181–193. Springer LNCS 6175, 2010.

[MMSP09] Vasco M. Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for weighted Boolean optimization. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT-2009, Swansea, UK*, pages 495–508. Springer LNCS 5584, 2009.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 39th Design Automation Conference, DAC'01*, pages 530–535, 2001.

[MS99] Joao P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Pedro Barahona and José Júlio Alferes, editors, *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence (EPIA-99)*, volume 1695 of *LNCS*, pages 62–74, 1999.

[MSG97] Bertrand Mazure, Lakhdar Saïs, and Éric Grégoire. Tabu search for SAT. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97, Providence/RI, USA*, pages 281–285. AAAI Press, 1997.

[MSK97] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97, Providence/RI, USA*, pages 321–326. AAAI Press, 1997.

[MSP08] Joao Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 408–413, Munich, Germany, 2008.

[MSS99] João P. Marques-Silva and Karem A. Sakallah. Graps: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[Nad02] A. Nadel. *Backtrack Search Algorithms for Propositional Logic Satisfiability: Review and innovations*. PhD thesis, Hebrew University of Jerusalem, 2002.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. Clone: Solving weighted max-sat in a reduced search space. In *Proceedings of the 20th Australian Conference on Artificial Intelligence, AI-2007, Gold Coast, Australia*, pages 223–233. Springer LNCS 4830, 2007.

[PJ10] Justyna Petke and Peter Jeavons. Local consistency and SAT-solvers. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP), Providence/RI, St. Andrews, Scotland, UK*, pages 398–413. Springer LNCS 6308, 2010.

[Pre93] Daniele Pretolani. Efficiency and stability of hypergraph SAT algorithms. In *Proceedings of the DIMACS Challenge II Workshop*, 1993.

[Pre07] Steven David Prestwich. Finding large cliques using SAT local search. In F. Benhamou, N. Jussien, and B. O'Sullivan, editors, *Trends in Constraint Programming*, chapter 15, pages 269–274. ISTE, 2007.

[Pre09] Steven David Prestwich. CNF encodings. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 75–97. IOS Press, 2009.

[RG07] Miquel Ramírez and Hector Geffner. Structural relaxations by variable renaming and their compilation for solving mincostsat. In *Proceedings of 13th International Conference on Principles and Practice of Constraint Programming, CP-2007, Providence, USA*, pages 605–619. Springer LNCS 4741, 2007.

[RM09] Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 695–733. IOS Press, 2009.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[Rya04] Lawrence Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[Sch89] Uwe Schöning. *Logic for Computer Scientists*, volume 8 of *Progress in Computer Science and Applied Logic*. Birkhäuser, 1989.

[SD96] B.M. Smith and M.E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155–181, 1996.

[SHR01] Thomas Stützle, Holger Hoos, and Andrea Roli. A review of the literature on local search algorithms for MAX-SAT. Technical report, AIDA-01-02, FG Intellektik, FB Informatik, TU Darmstadt, Germany, 2001.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proceedings of the 11th International Conference on*

Principles and Practice of Constraint Programming (CP), Sitges, Spain, pages 827–831. Springer LNCS 3709, 2005.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94, Seattle/WA, USA*, pages 337–343. AAAI Press, 1994.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI'92, San Jose/CA, USA*, pages 440–446. AAAI Press, 1992.

[Soo09] Mate Soos. Extending sat solvers to cryptographic problems. In *BOOKTITLE = Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT-2009, Swansea, UK*, pages 244–257. Springer LNCS 5584, 2009.

[SW02] John Slaney and Toby Walsh. Phase transition behavior: from decision to optimization. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing, SAT-2002*, 2002.

[SZ04] Haiou Shen and Hantao Zhang. Study of lower bound functions for max-2-sat. In *Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence, San Jose, California, USA*, pages 185–190. AAAI Press / The MIT Press, 2004.

[TH05] Dave A. D. Tompkins and Holger H. Hoos. Ubcsat: An implementation and experimentation environment for sls algorithms for sat and max-sat. In *7th International Conference on Theory and Applications of Satisfiability Testing, SAT-2004, Vancouver, BC, Canada*, pages 306–320. Springer LNCS 3542, 2005.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[VB01] Miroslav N. Velev and Randal E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proceedings of the 38th Design Automation Conference, DAC-2001, Las Vegas/NV, USA*, pages 226–231, 2001.

[Wal00] Toby Walsh. SAT v CSP. In *Proceedings of the 6th International Conference on Principles of Constraint Programming (CP), Singapore*, pages 441–456. Springer LNCS 1894, 2000.

[WvM98] Joost P. Warners and Hans van Maaren. A two-phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters*, 23:81–88, 1998.

[XZ04] Zhao Xing and Weixiong Zhang. Efficient strategies for (weighted) maximum satisfiability. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, CP-2004, Toronto, Canada*, pages 690–705. Springer LNCS 3258, 2004.

[XZ05] Zhao Xing and Weixiong Zhang. An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence*, 164(2):47–80, 2005.

[Yan94] Mihalis Yannakakis. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17:475–502, 1994.

[Zha97] Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction (CADE-97)*, pages 272–275, 1997.

[Zha03] Lintao Zhang. *Searching for truth: techniques for satisfiability of Boolean formulas*. PhD thesis, Department of Electrical Engineering. Princeton University., 2003.

[ZLS04] Hantao Zhang, Dapeng Li, and Haiou Shen. A sat based scheduler for tournament schedules. In *7th International Conference on Theory and Applications of Satisfiability Testing, SAT-2004, Vancouver, BC, Canada*. Springer LNCS 3542, 2004.

[ZM88] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *In Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, pages 155–160, 1988.

[ZM02] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *18th International Conference on Automated Deduction, CADE-18, Copenhagen, Denmark*, pages 295–313. Springer, LNCS 2392, 2002.

[ZS96] Hantao Zhang and Mark E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics, AI-MATH'96*, Fort Lauderdale (Florida USA), 1996.

[ZSM03] Hantao Zhang, Haiou Shen, and Felip Manya. Exact algorithms for MAX-SAT. *Electronic Notes in Theoretical Computer Science*, 86(1), 2003.