TÉCNICO LISBOA

Distributed Solver for Maximum Satisfiability

Miguel Ângelo da Terra Neves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Professor Vasco Miguel Gomes Nunes Manquinho
Professor Maria Inês Camarate de Campos Lynce de Faria

Examination Committee

Chairperson: Professor Ernesto José Marques Morgado
Supervisor: Professor Vasco Miguel Gomes Nunes Manquinho
Member of the Committee: Professor Luís Jorge Brás Monteiro Guerra e Silva

October 2014

Acknowledgments

First of all I would like to thank my parents Ana Maria and Nascimento for providing everything I needed in order to go through a higher education, in the course that I always wished for, and helping me throughout the first steps of achieving a fulfilling career. Without them I would never have made it this far.

I want to thank my great friends, in particular Daniela Rosa, André Silva, Dima Orban, Guilherme Santos and Catarina Cardoso, for all the support, always being there in my time of need and for making me feel proud of what I have accomplished so far.

I also want to thank my supervisors Professor Vasco Manquinho and Professor Inês Lynce for all the dedicated feedback and support given throughout the development of this master thesis, all the help with finding finance support for the project and with my application for a PhD scholarship. I have learned a lot this last year, in particular about research and writing a nicely structured scientific document.

Finally, I would like to thank Fundação da Ciência e Tecnologia for supporting this research work through project ASPEN (PTDC/EIA-CCO/110921/2009).

Resumo

Satisfação Máxima (MaxSAT) é uma versão de optimização do problema de *Satisfação Booleana* (SAT). Muitos problemas do mundo real podem ser reduzidos a resolver uma instância de MaxSAT, incluindo problemas de interesse industrial. Portanto, MaxSAT é um problema importante para se resolver e, nos anos mais recentes, muitos algoritmos eficientes têm sido desenvolvidos, com o propósito de resolver este problema, e analisados. Alguns destes algoritmos são paralelos, mas nenhum algoritmo distribuído foi desenvolvido até agora.

Esta dissertação analiza as abordagens existentes para resolver MaxSAT e propõe dois algoritmos distribuídos para um aglomerado de processadores sobre o controlo do utilizador. Estes dois algoritmos foram implementados e depois testados, estudados e comparados com um dos solvers sequenciais de MaxSAT mais eficazes que existem na actualidade. O primeiro algoritmo é muito semelhante a um algoritmo paralelo de sucesso já existente para MaxSAT, e o segundo é uma adaptação de uma abordagem que mostrou ser bastante eficaz na resolução sequencial, paralela e distribuída de SAT.

Palavras-chave: Satisfação Booleana (SAT), Boolean Optimization, Satisfação Máxima (MaxSAT), Procura Distribuída, Divisão do Espaço de Procura, Caminhos Norteadores com Lookahead.

Abstract

Maximum Satisfiability (MaxSAT) is an optimization version of *Boolean Satisfiability* (SAT). Many real world problems can be reduced to solving a MaxSAT instance, including problems of industrial interest. Therefore, MaxSAT is an important problem to solve and, in the recent years, many efficient algorithms have been developed, in order to tackle the challenge, and analyzed. Some of these algorithms are parallel, but no distributed algorithm has been developed so far.

This paper analyzes the existing approaches for solving MaxSAT and proposes two distributed algorithms for a cluster of processors under the user's control. These two algorithms were implemented and then experimented, studied and compared with one of the most effective sequential MaxSAT solvers that exist nowadays. The first algorithm is very similar to an existing successful parallel algorithm for MaxSAT, and the second one is an adaptation of an approach that proved to be very effective in sequential, parallel and distributed SAT solving.

Keywords: Boolean Satisfiability (SAT), Boolean Optimization, Maximum Satisfiability (MaxSAT), Distributed Search, Search Space Splitting, Guiding Paths with Lookahead.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Suppose that we are in charge of designing a cover for a rock band's new album. In order to fill the deadline, we have to choose among the 4 different cover sketches that we proposed to the band. The problem is, there was not a single sketch that pleased every band member, and we intend to choose the sketch that will satisfy the majority of them. The band is formed by a singer, a lead guitarist, a rhythm guitarist, a bassist and a drummer. The reactions to each sketch were as follows: sketch 1 was disliked by the singer, sketch 2 was disliked by the guitarists, sketch 3 was disliked by the drummer and the bassist and sketch 4 was disliked by the singer. Also, the producers require that we satisfy the singer and the drummer due to their troublesome personalities.

This is an example of a logic optimization problem that could be modeled as follows: s, lg, rg, b, d are Boolean variables that encode, respectively, if the singer, lead guitarist, rhythm guitarist, bassist and drummer are satisfied (for example, s is 1 if the singer is satisfied and 0 otherwise). $x_1, ..., x_4$ are Boolean variables that encode which cover we chose to produce (x_i is 1 if we chose sketch i and 0 otherwise). $s \wedge d$ is a conjunction expressing that the singer and the drummer must be pleased with our choice. $\phi = (\neg x_1 \vee \neg s) \wedge (\neg x_2 \vee \neg lg) \wedge (\neg x_2 \vee \neg rg) \wedge (\neg x_3 \vee \neg d) \wedge (\neg x_3 \vee \neg b) \wedge (\neg x_4 \vee \neg s)$ is a conjunction expressing the preferences of each band member. There is also the implicit constraint that exactly one sketch must be chosen among the 4. This property is expressed by the conjunction $\psi = \bigwedge_{1 \leq i < j \leq 4} (\neg x_i \vee \neg x_j)$ together with the clause $(x_1 \vee x_2 \vee x_3 \vee x_4)$.

The goal is to determine an assignment to the variables that satisfies the conjunction $s \wedge d \wedge \phi \wedge (x_1 \vee x_2 \vee x_3 \vee x_4) \wedge \psi$ and maximizes the number of formulas in the set $\{(s), (lg), (rg), (b), (d)\}$ that are satisfied. This is an instance of a counterpart of the optimization problem known as *Maximum Satisfiability* (MaxSAT), which is known to be very hard to solve efficiently. The solution for this instance is $s = d = b = x_2 = 1$, $rg = lg = x_1 = x_3 = x_4 = 0$.

Many real-world problems can be reduced to the MaxSAT framework. It has been shown to be useful in applications related to Computational Biology (Strickland et al. [2005]), Routing (Xu et al. [2003]), Circuit Debugging (Mangassarian et al. [2007]), Software Debugging (Jose and Majumdar [2011]), among others.

1.1 Why Distributed MaxSAT?

Much work has been done on algorithms for sequential MaxSAT during the last decade. Therefore, it has become harder to improve sequential algorithms. Besides, nowadays the computational power is increasing due to the predominance of multicore architectures instead of higher frequency CPUs. For this reason, recent work on MaxSAT solving has been done on the design of parallel solvers.

There is still room to improve the current state-of-the-art parallel MaxSAT algorithms, but the parallel approach also has its drawbacks. All the cores of a typical computer access primary memory through the same BUS, and no two cores can use the BUS at the same time. Therefore, as the number of cores increases, so does contention on the access to primary memory, hindering the scalability of parallel algorithms. Machines with a BUS per core could be built, but this would further increase its cost. Moreover, as cores are added to a machine, it becomes much more costly.

Positive results have been obtained through recent work on parallel MaxSAT algorithms (Martins [2013]). However, for the reasons mentioned previously, parallel algorithms hardly use more than 8 threads. Therefore, it is interesting to conduct research on distributed algorithms in order to further improve MaxSAT solving.

1.2 Contributions

This document provides a succinct description of the algorithms that have been designed so far for solving MaxSAT, both sequential and parallel approaches. Note that we are only interested in complete algorithms, that is, algorithms that always return the optimum solution.

Moreover, this document introduces two new algorithms that were implemented and compared in a first attempt to develop a distributed MaxSAT solver. These algorithms are strongly related to previous approaches proposed for parallel SAT and parallel MaxSAT solving which showed positive results. The first algorithm splits the search space by assigning different upper bound values of the optimum solution to different processes, very similar to the strategy used in the PWBO (Martins [2013]) parallel solver. The second approach is based on choosing a subset of the problem's variables and dividing the possible combinations of values for those variables among the processes.

1.3 Document Organization

Chapter 2 starts by introducing some essential definitions, notations and concepts that are used throughout the document.

In chapter 3, previous work on solving MaxSAT is described. We start by presenting sequential algorithms for MaxSAT in section 3.1. Sections 3.1.1 to 3.1.3 focus on algorithms that rely on iteratively invoking a SAT solver. Next, Branch and Bound approaches for MaxSAT are described in section 3.1.4. Parallel algorithms are described in section 3.2. These algorithms fall into two different categories:

Portfolio (section 3.2.1) and Search Space Splitting (section 3.2.2). Heuristics for clause sharing in parallel algorithms are introduced in section 3.2.3.

In chapter 4 the two distributed algorithms for MaxSAT are introduced. Firstly, the search space splitting algorithm based on upper bounds is described in section 4.1. Then, section 4.2 describes the algorithm that divides the problem instance based on the variables.

In chapter 5 the experimental results obtained for both distributed algorithms are presented and analyzed. The results for the algorithm introduced in section 4.1 are presented in section 5.1. Next, the results for the algorithm described in section 4.2 are presented and compared with the results obtained for the first algorithm in section 5.2.

Finally, chapter 6 concludes this dissertation.

Chapter 2

Preliminaries

In this chapter some definitions, notations and concepts that will be used throughout the rest of this document are introduced. The Boolean Satisfiability problem is described in section 2.1 and Maximum Satisfiability is described in section 2.2.

2.1 The Boolean Satisfiability (SAT) Problem

Before detailing the SAT problem, the following definitions are necessary.

Definition 2.1.1. A *literal* is either a Boolean variable (*positive literal*) or the negation of a variable (*negative literal*).

Definition 2.1.2. A *clause* is a disjunction of literals.

Definition 2.1.3. A *unit clause* is a clause with exactly one literal.

Definition 2.1.4. A propositional logic formula in the *conjunctive normal form* (CNF) is a conjunction of clauses.

Definition 2.1.5. Given a propositional logic formula ϕ, an *assignment* is a partial mapping $\nu : X \rightarrow \{0, 1\}$ where X is the set of all variables in ϕ.

Definition 2.1.6. Given an assignment $\nu : X \rightarrow \{0, 1\}$ and a variable $x \in X$, the positive literal x is *satisfied* by ν if and only if $\nu(x) = 1$ and the negative literal $\neg x$ is *satisfied* by ν if and only if $\nu(x) = 0$; a clause is *satisfied* by ν if and only if at least one of its literals is *satisfied* by ν; a propositional logic formula in CNF is *satisfied* by ν if and only if all of its clauses are *satisfied* by ν.

Definition 2.1.7. Given a propositional logic formula in CNF ϕ and an assignment ν, ν is a *model* of ϕ if and only if ν satisfies ϕ.

Definition 2.1.8. A propositional logic formula in CNF ϕ is *unsatisfiable* if and only if no assignment ν exists such that ν satisfies all the clauses in ϕ.

Any propositional logic formula can be converted to an equivalent CNF formula. Therefore, a SAT instance can be represented as a conjunction of clauses ϕ. A solution to the SAT instance ϕ is an assignment ν on the variables of ϕ that satisfies ϕ.

Notation 2.1.1. Let $c_1, ..., c_n$ be clauses and $\phi = c_1 \wedge ... \wedge c_n$ a propositional logic formula in CNF. The set of clauses $\{c_1, ..., c_n\}$ is an equivalent notation for ϕ.

Throughout this document, a CNF formula will be represented by a set of clauses as introduced above.

Example 2.1.1. Consider the following propositional logic formula in CNF $\phi = \{(x_1 \vee x_2 \vee \neg x_3), (\neg x_2 \vee \neg x_3)\}$. x_1 and x_2 are positive literals; $\neg x_2$ and $\neg x_3$ are negative literals. The assignment $\nu_1 = \{(x_1, 1), (x_2, 0), (x_3, 1)\}$ satisfies the formula ϕ, but the assignment $\nu_2 = \{(x_1, 0), (x_2, 1), (x_3, 1)\}$ does not.

Given an unsatisfiable set of clauses ϕ, it may be required for the SAT solver to provide a more detailed explanation for the unsatisfiability of ϕ.

Definition 2.1.9. Let ϕ be an unsatisfiable set of clauses. A subset of clauses $\phi_C \subseteq \phi$ is an *unsatisfiable core* if ϕ_C is also unsatisfiable. ϕ_C is a *minimal unsatisfiable core* if all the subsets $\psi \subset \phi_C$ are satisfiable.

Example 2.1.2. Consider the unsatisfiable set of clauses $\phi = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2), (x_3)\}$. ϕ is an unsatisfiable core, but it is not minimal. The set $\phi_C = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2)\}$ is a minimal unsatisfiable core.

Conflict Driven Clause Learning (CDCL) SAT solvers (Marques-Silva et al. [2009]) are among the most effective modern SAT solvers. These solvers apply a depth-first backtrack search where at each branching step a variable is chosen and a truth value is assigned to it. Each time a clause becomes unsatisfied (referred to as a conflict), CDCL SAT solvers backtrack up to a variable for which both truth values have not yet been tested and no clause is unsatisfied. Additionally, CDCL SAT solvers learn new clauses from conflicts through a process referred to as *conflict analysis* (Marques-Silva and Sakallah [1996], Zhang et al. [2001]). It is widely known that learning new clauses that occur during the search can speed up the search process of Boolean solvers by pruning the search space.

Sometimes it is useful to invoke a SAT solver on the same instance multiple times but with different preset assignments to specific variables. This is achieved through the use of assumptions. Assumptions can be passed to a SAT solver when invoked. An assumption is a literal that only holds on that specific invocation of the SAT solver. Therefore, an assumption is very similar to a unit clause, but unlike the latter, assumptions are discarded after the search is finished. When the SAT solver learns a clause that is conflicting with the given assumptions, the search terminates returning that the formula is unsatisfiable. It is not possible to remove clauses from the solver between invocations, because removing a clause c after an invocation would imply removing also the learned clauses that were derived from c in order for the SAT solver to remain in a valid state. Therefore, assumptions should be used when one needs to enforce certain variable assignments only on a specific invocation to the SAT solver.

2.2 The Maximum Satisfiability (MaxSAT) Problem

A MaxSAT instance can be encoded in the same way as a SAT instance, but in the MaxSAT case, given a set of clauses ϕ, the goal is to determine an assignment ν over the variables in ϕ that maximizes the number of satisfied clauses in ϕ. Such an assignment is defined as an *optimum solution* (or *optimum assignment*). This version of MaxSAT is referred to as *unweighted MaxSAT*. We can also look at MaxSAT as a minimization problem instead of maximization. In this case, the goal is to minimize the number of unsatisfied clauses. Throughout the rest of this document it is assumed that MaxSAT is defined as a minimization problem.

Example 2.2.1. Consider the unweighted MaxSAT instance encoded by the set of clauses $\phi = \{(x_1 \vee x_2), (\neg x_1 \vee x_2), (\neg x_2)\}$. It is not hard to verify that there exists no assignment that satisfies all clauses in ϕ. On the other hand, there are several assignments that unsatisfy only one of the clauses in ϕ. For example, the assignment $\nu = \{(x_1, 1), (x_2, 1)\}$ unsatisfies only the last clause. Therefore, ν is an optimum solution since it is not possible to satisfy all clauses.

Before detailing other variations of the MaxSAT problem, we need to introduce the following notation and definition.

Notation 2.2.1. Let ϕ be a set of clauses, X the set of all variables in ϕ, ν an assignment over ϕ and c a clause such that $c \in \phi$. We denote as $\nu(c)$ the value of clause c under assignment ν, in other words:

- $\nu(c) = 0$ if and only if ν unsatisfies c;
- $\nu(c) = 1$ otherwise.

Definition 2.2.1. A *weighted clause* is a pair (c, w) such that c is a clause, w is a positive integer weight and $w \geq 1$. w is the *cost* of unsatisfying c.

In *weighted MaxSAT*, each clause c_i has a cost w_i assigned to it. Therefore, an instance of weighted MaxSAT is encoded as a set of weighted clauses $\phi = \{(c_1, w_1), ..., (c_N, w_N)\}$ where N is the total number of weighted clauses in ϕ. An optimum solution for a weighted MaxSAT instance is an assignment ν over the variables in ϕ that minimizes the sum of the costs of the unsatisfied clauses, that is, ν minimizes the following expression:

$$\sum_{i=1}^{N} w_i \cdot (1 - \nu(c_i)) \tag{2.1}$$

Example 2.2.2. Consider the weighted MaxSAT instance encoded by the set of weighted clauses $\phi_w = \{(x_1 \vee x_2, 2), (\neg x_1 \vee x_2, 1), (\neg x_2, 3)\}$, which is similar to the instance ϕ in example 2.2.1, except that the clauses now have costs. The assignment ν in example 2.2.1 is an optimum solution for ϕ, but it is not an optimum solution for ϕ_w, because the cost of the unsatisfied clause is 3 and it is possible to obtain a better solution for ϕ_w. The optimum solution for instance ϕ_w is the assignment $\nu_{opt} = \{(x_1, 1), (x_2, 0)\}$, since it unsatisfies only the second clause, which has a cost of 1, and no better solution exists.

Sometimes, in many real world applications, it is required that some of the clauses must be satisfied. Such a variation of MaxSAT is encoded with two sets of clauses, a set ϕ_H of *hard clauses* and a set ϕ_S of *soft clauses*, where the hard clauses are the ones that must be satisfied. This variation of MaxSAT is referred to as *partial MaxSAT* (Cha et al. [1997]) and an optimum solution to such an instance is an assignment ν over the variables in ϕ_H and ϕ_S that satisfies all the clauses in ϕ_H and minimizes the number of unsatisfied clauses in ϕ_S.

With the addition of hard clauses to the MaxSAT problem, it becomes relevant to distinguish between valid and invalid assignments. It also becomes possible for a MaxSAT instance to be unsatisfiable.

Definition 2.2.2. Let (ϕ_H, ϕ_S) be an instance of partial MaxSAT. An assignment ν is *valid* for (ϕ_H, ϕ_S) if, and only if, ν satisfies all of the clauses in ϕ_H. Otherwise, ν is *invalid* for (ϕ_H, ϕ_S).

Definition 2.2.3. Let (ϕ_H, ϕ_S) be a partial MaxSAT instance. (ϕ_H, ϕ_S) is *unsatisfiable* if, and only if, ϕ_H is unsatisfiable.

Example 2.2.3. Consider the partial MaxSAT instance (ϕ_H, ϕ_S), where $\phi_H = \{(\neg x_1)\}$ and $\phi_S = \{(x_1 \vee x_2), (\neg x_1 \vee x_2), (\neg x_2)\}$. Consider also the set of clauses ϕ from example 2.2.1 and notice that $\phi_S = \phi$. The assignment ν from example 2.2.1 is an optimum solution for the unweighted MaxSAT instance ϕ, but it is not even a valid assignment for the partial MaxSAT instance (ϕ_H, ϕ_S). On the other hand, the assignment $\nu_{opt} = \{(x_1, 0), (x_2, 0)\}$ is both valid and an optimum solution for (ϕ_H, ϕ_S).

Example 2.2.4. Consider the partial MaxSAT instance (ϕ_H, ϕ_S), where $\phi_H = \{(x_1), (\neg x_1)\}$ and $\phi_S = \{(x_1 \vee x_2), (\neg x_1 \vee x_2), (\neg x_2)\}$. (ϕ_H, ϕ_S) is unsatisfiable because ϕ_H is unsatisfiable.

There is yet another version of MaxSAT named *weighted partial MaxSAT*, which is similar to partial MaxSAT. The difference is that the set of soft clauses is a set of weighted clauses $\phi_S = \{(c_1, w_1), ..., (c_N, w_N)\}$. Therefore, an optimum solution to an instance of weighted partial MaxSAT is an assignment ν that satisfies all the clauses in ϕ_H and minimizes the total weight of unsatisfied clauses (2.1).

2.2.1 Cardinality Constraints

Several state-of-the-art MaxSAT solvers apply algorithms that iteratively call a SAT solver with a relaxed version of the problem instance. Such algorithms use constraints that restrict how many literals of a given set can be satisfied simultaneously. These constraints are called *cardinality constraints*. However, state-of-the-art SAT solvers do not offer native support for cardinality constraints. Therefore, it is necessary to encode those constraints into CNF.

Firstly, cardinality constraints must be defined.

Definition 2.2.4. Let $l_1, ..., l_N$ be literals and k a non negative integer. A *cardinality constraint* is a constraint with the following form:

$$\sum_{i=1}^{N} l_i \leq k. \tag{2.2}$$

Cardinality constraints as defined above are also referred to as *at-most-k constraints*. Note that constraints of the form $\sum_{i=1}^{N} l_i \geq k$, referred to as *at-least-k constraints*, are equivalent to $\sum_{i=1}^{N} \neg l_i \leq N - k$.

Notation 2.2.2. Given an at-most-k constraint C, we denote as $[C]_{CNF}$ a CNF encoding for C.

A CNF encoding for at-most-k constraints must be sound and complete. Given an at-most-k constraint C and a CNF encoding $[C]_{CNF}$ for that constraint, the set of all assignments that satisfy C must be exactly the same that satisfy $[C]_{CNF}$. A summary of encodings for cardinality constraints is presented in appendix A. The definition of satisfaction for at-most-k constraints is the following.

Definition 2.2.5. Given an integer k, an at-most-k constraint C and an assignment ν, we say that ν *satisfies* C if no more than k of the literals in C are satisfied by ν.

Example 2.2.5. Consider the at-most-k constraint $x_1 + \neg x_2 + x_3 \leq 2$ and the assignments $\nu_1 = \{(x_1, 1), (x_2, 0), (x_3, 1)\}$, $\nu_2 = \{(x_1, 1), (x_2, 1), (x_3, 1)\}$ and $\nu_3 = \{(x_1, 0), (x_2, 1), (x_3, 1)\}$. The assignment ν_1 does not satisfy the at-most-k constraint because ν_1 satisfies more than two literals. On the other hand, the assignments ν_2 and ν_3 satisfy the constraint because ν_2 and ν_3 do not satisfy more than two literals (ν_2 satisfies two literals and ν_3 satisfies one).

The following definition is a more general case of a cardinality constraint.

Definition 2.2.6. Let $l_1, ..., l_N$ be literals, $a_1, ..., a_n$ be positive integer coefficients and k a non negative integer. A *pseudo-Boolean constraint* is a constraint with the following form:

$$\sum_{i=1}^{N} a_i l_i \leq k. \tag{2.3}$$

Pseudo-Boolean constraints are useful when solving instances of the weighted variants of MaxSAT, but those variants are not the main focus of this project. For the interested reader, a series of encodings for pseudo-Boolean constraints can be found in the literature (Eén and Sörensson [2006], Warners [1998], Bailleux et al. [2006]).

Chapter 3

Related Work

This chapter presents previous work for solving the MaxSAT problem. Firstly, the sequential algorithms are presented. Next, we focus on parallel algorithms.

3.1 Sequential MaxSAT

We start by presenting the iterative family of algorithms, which includes the Linear Search, Binary Search and Core Guided algorithms. Next, the Branch and Bound procedure is described.

In the presented algorithms is assumed that ϕ_H is satisfiable. Note that the satisfiability of ϕ_H could be checked in a pre processing phase by invoking a SAT solver on ϕ_H.

3.1.1 Linear Search Algorithms

The first Linear Search algorithm is the *Linear Search Sat-Unsat* (An et al. [2012], Le Berre and Parrain [2010]) algorithm. This algorithm iteratively invokes a SAT solver on a relaxed satisfiable version of the MaxSAT instance until an unsatisfiable version is found. The relaxed version is constructed by adding *relaxation variables* $r_1, ..., r_K$ to each of the K clauses in ϕ_S. Then, an upper bound μ on the number of unsatisfied clauses of the optimal assignment is computed (a trivial possibility is to initialize μ as $|\phi_S| + 1$). The *working formula* ϕ_W is the union of ϕ_H with ϕ_S augmented with fresh relaxation variables and $[\sum_{i=1}^{K} r_i \leq \mu - 1]_{CNF}$. When the SAT solver returns that ϕ_W is satisfiable, the upper bound μ is refined and ϕ_W is updated accordingly. If the SAT solver returns unsatisfiable, then the last assignment returned by the SAT solver, excluding the relaxation variables, is the optimum solution and the optimal number of unsatisfied clauses is the current μ (figure 3.1).

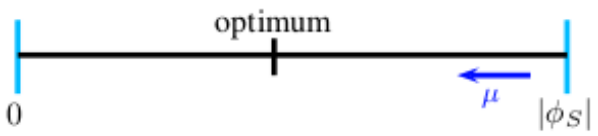

Figure 3.1: Linear search.

Algorithm 1: *Linear Search Sat-Unsat* algorithm for partial MaxSAT (An et al. [2012])

Input: (ϕ_H, ϕ_S)
1 $(\phi_W, R) \leftarrow$ `Relax(`ϕ_S`)`
2 $\phi_W \leftarrow \phi_W \cup \phi_H$
3 $\mu \leftarrow |\phi_S| + 1$
4 $status \leftarrow SAT$
5 $\nu \leftarrow nil$
6 **while** $status = SAT$ **do**
7 $\quad (status, \nu) \leftarrow$ `SAT(`$\phi_W \cup [\sum_{i=1}^{K} r_i \leq \mu - 1]_{CNF}$`)`
8 $\quad$ **if** $status = SAT$ **then** `/* refine upper bound */`
9 $\quad\quad \mu \leftarrow |\{r_i \in R : \nu(r_i) = 1\}|$
10 $\quad$ **end**
11 **end**
12 $\nu \leftarrow$ `RemoveVariables(`ν`,` R`)`
13 **return** ν

The pseudo-code for the Linear Search Sat-Unsat algorithm is presented in algorithm 1. In line 8, the upper bound is refined by setting μ to the number of relaxation variables that were assigned the truth value 1 in the last *assignment* returned by the SAT solver. The functions in the algorithm work as follows:

- *Relax*(ϕ) adds fresh relaxation variables to the clauses in ϕ. The relaxed formula and the set of new relaxation variables are returned;
- *SAT*(ϕ) checks if the set of clauses ϕ is satisfiable. If so, an assignment that satisfies ϕ is returned;
- *RemoveVariables*(ν, X) returns the assignment ν without the variables in X.

Example 3.1.1. Consider the set of clauses $\phi = \{(x_1), (\neg x_1 \vee x_2), (\neg x_3)\}$. The result of *Relax*($\phi$) is the set of clauses $\{(x_1 \vee r_1), (\neg x_1 \vee x_2 \vee r_2), (\neg x_3 \vee r_3)\}$.

The purpose of relaxing the soft clauses in algorithm 1 is to allow those clauses to be unsatisfied by the SAT solver. For example, by assigning the truth value 1 to r_i, it means that the original soft clause c_i may be unsatisfied. The number of soft clauses that may be unsatisfied is restricted by the at-most-k constraint imposed on the relaxation variables. The invocation of the SAT solver in line 6 can be interpreted as asking the following question: "Does an assignment exist such that all the hard clauses are satisfied and at most $\mu - 1$ of the soft clauses are unsatisfied?".

Example 3.1.2. Consider the partial MaxSAT instance (ϕ_H, ϕ_S) where $\phi_S = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2)\}$. After step 2 in algorithm 1, we have the following working formula:

$$\phi_W = \phi_H \cup \{(x_1 \vee x_2 \vee r_1), (\neg x_1 \vee r_2), (\neg x_2 \vee r_3)\} \tag{3.1}$$

The initial upper bound is $\mu = 4$. The SAT solver is invoked with the propositional formula:

$$\phi_W \cup [r_1 + r_2 + r_3 \leq 3]_{CNF} \tag{3.2}$$

Suppose that the SAT solver returns the assignment $\nu = \{(x_1, 0), (x_2, 0), (r_1, 1), (r_2, 0), (r_3, 0)\}$. Since

only one of the relaxation variables is assigned 1 in the assignment ν, the algorithm assigns 1 to μ as the new upper bound.

The SAT solver is then invoked with the propositional formula:

$$\phi_W \cup [r_1 + r_2 + r_3 \leq 0]_{CNF} \tag{3.3}$$

The SAT solver returns that the formula is unsatisfiable, so the algorithm returns assignment ν, which is an optimum solution. It is easy to verify that ν unsatisfies exactly $\mu = 1$ of the soft clauses in ϕ_S.

Algorithm 1 as it is presented only works for the unweighted case. The version of the algorithm proposed by Le Berre and Parrain [2010] solves instances of weighted partial MaxSAT. To adapt algorithm 1 to weighted partial MaxSAT only two changes are needed. Firstly, a pseudo-Boolean constraint is imposed on the relaxation variables instead of a cardinality constraint, where each coefficient a_i, that is associated to a relaxation variable r_i, is the weight of the corresponding soft clause.

Example 3.1.3. Consider the following set of relaxed soft clauses $\phi = \{(x_1 \vee x_2 \vee r_1), (\neg x_1 \vee r_2), (\neg x_2 \vee r_3)\}$ with weights w_1, w_2 and w_3 respectively. The following equation is an example of a pseudo-Boolean constraint on the relaxation variables:

$$w_1 r_1 + w_2 r_2 + w_3 r_3 \leq k \tag{3.4}$$

The second change is in the refinement of the upper bound μ. In the weighted case, μ is an upper bound of the sum of the weights of the unsatisfied soft clauses of the optimum solution. μ is refined by assigning to it the sum of the weights associated with the relaxation variables that were assigned the truth value 1 in the assignment returned by the SAT solver.

3.1.2 Binary Search Algorithms

The traditional Binary Search (An et al. [2011]) algorithm refines both a lower bound and an upper bound. The algorithm starts by relaxing the soft clauses, in the same way as algorithm 1, and computing the initial lower and upper bounds λ and μ (a trivial possibility for initializing λ is -1). Then, the algorithm proceeds to iteratively invoke the SAT solver on a working formula like the one in algorithm 1, except that the at-most-k constraint on the relaxation variables has k set to the *middle point* between λ and μ. If the solver returns that the working formula is unsatisfiable, the lower bound λ is refined, but if the working formula is satisfiable, the upper bound μ is refined instead. The algorithm ends when the condition $\lambda + 1 < \mu$ does not hold anymore, and the optimum solution is the last assignment returned by the SAT solver.

The pseudo-code for the Binary Search algorithm is presented in algorithm 2. The upper bound μ is refined in line 8 in the same way as in algorithm 1. The lower bound λ is refined in line 10 by setting λ to the middle value ρ, because if there is no assignment that unsatisfies at most ρ soft clauses, then the number of soft clauses unsatisfied by the optimum assignment must be larger than ρ.

Algorithm 2: *Binary Search* (An et al. [2011]) algorithm for partial MaxSAT

Input: (ϕ_H, ϕ_S)
1 $(\phi_W, R) \leftarrow$ `Relax(`ϕ_S`)`
2 $\phi_W \leftarrow \phi_W \cup \phi_H$
3 $\lambda \leftarrow -1$
4 $\mu \leftarrow |\phi_S| + 1$
5 $\nu \leftarrow nil$
6 **while** $\lambda + 1 < \mu$ **do**
7 $\quad \rho \leftarrow \lfloor \frac{\lambda+\mu}{2} \rfloor$
8 $\quad (status, \nu) \leftarrow$ `SAT(`$\phi_W \cup [\sum_{i=1}^{K} r_i \leq \rho]_{CNF}$`)`
9 $\quad$ **if** $status = SAT$ **then** `/* refine upper bound */`
10 $\quad\quad \mu \leftarrow |\{r_i \in R : \nu(r_i) = 1\}|$
11 $\quad$ **else** `/* refine lower bound */`
12 $\quad\quad \lambda \leftarrow \lfloor \frac{\lambda+\mu}{2} \rfloor$
13 $\quad$ **end**
14 **end**
15 $\nu \leftarrow$ `RemoveVariables(`ν, R`)`
16 **return** ν

Example 3.1.4. Consider the partial MaxSAT instance (ϕ_H, ϕ_S) where $\phi_S = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2)\}$, which is the same instance given in example 3.1.2.

The initial lower and upper bounds are $\lambda = -1$ and $\mu = 4$. The middle value is $\rho = 1$. The SAT solver is invoked with the propositional formula:

$$\phi_W \cup [r_1 + r_2 + r_3 \leq 1]_{CNF} \tag{3.5}$$

Suppose that the SAT solver returns the assignment $\nu = \{(x_1, 0), (x_2, 0), (r_1, 0), (r_2, 1), (r_3, 0)\}$. Since the solver returned that the propositional formula is satisfiable, the upper bound μ must be refined. The algorithm assigns 1 to μ as the new upper bound because only one of the relaxation variables is assigned 1 in the assignment ν.

The new middle value is $\rho = 0$. The SAT solver is invoked with the following new propositional formula:

$$\phi_W \cup [r_1 + r_2 + r_3 \leq 0]_{CNF} \tag{3.6}$$

The SAT solver returns that the formula is unsatisfiable, so the lower bound λ must be refined. The algorithm assigns 0 to λ.

At this time, the condition $\lambda + 1 < \mu$ does not hold anymore. Therefore the assignment ν is returned as the optimum solution. As in example 3.1.2, ν unsatisfies exactly 1 soft clause in ϕ_S.

An et al. [2011] proposed a variant of algorithm 2 that alternates between Binary Search and Linear Search. This variant stores the mode that the algorithm is in, and this mode can be one of the aforementioned types of search. Also, this mode is swapped after each iteration of the main cycle. At each iteration, if the mode is set to Binary Search, then the algorithm acts as algorithm 2 for that iteration, but if the mode is set to Linear Search, then the algorithm acts as algorithm 1. The rationale for this variant is that Binary Search is better than Linear Search in some instances, but others are solved faster with

Algorithm 3: *Bit-Based Search* algorithm for partial MaxSAT

Input: (ϕ_H, ϕ_S)
$(\phi_W, R) \leftarrow \texttt{Relax}(\phi_S)$
$\phi_W \leftarrow \phi_W \cup \phi_H$
$k \leftarrow \lfloor \log_2 |\phi_S| \rfloor$
$i \leftarrow k$
$\eta \leftarrow 2^i$
$\nu \leftarrow nil$
while $i \geq 0$ **do**
  $(status, \nu) \leftarrow \texttt{SAT}(\phi_W \cup [\sum_{i=1}^{K} r_i \leq \eta]_{CNF})$
  if $status = SAT$ **then**
    let $s_k, ..., s_0$ be the bit representation of $|\{r_i \in R : \nu(r_i) = 1\}|$
    $i \leftarrow \text{max}(\{j : j < i \wedge s_j = 1\} \cup \{-1\})$
    $\eta \leftarrow \sum_{j=i}^{k} 2^j \times s_j$
  else
    $i \leftarrow i - 1$
    $\eta \leftarrow \eta + 2^i$
  end
end
$\nu \leftarrow \texttt{RemoveVariables}(\nu, R)$
return ν

the Linear Search algorithm.

Another algorithm for solving MaxSAT with Binary Search is the *Bit-Based Search* algorithm. This algorithm computes the number of unsatisfied clauses in the optimum solution bit by bit, starting with the most significant bit. The algorithm starts by computing the minimum number of bits to represent $|\phi_S|$, which is the highest number of soft clauses that can be unsatisfied for that particular instance. The algorithm then proceeds to compute each bit b_i, $i < |\phi_S|$, of the optimum number η of unsatisfied clauses, terminating when all bits have been computed.

The pseudo-code for Bit-Based Search (Codish et al. [2008], Giunchiglia and Maratea [2007]) is presented in algorithm 3. When the SAT solver returns that the given propositional formula is unsatisfiable, it means that the assignment corresponding to the optimum solution must unsatisfy more than η soft clauses, so bit i of η is left at 1, and the algorithm increases η by setting bit $i - 1$ to 1[1]. If the SAT solver returns that the propositional formula is satisfiable instead, then the algorithm computes the bit representation, $s_k, ..., s_0$, of the number of relaxation variables that were assigned the truth value 1 by the SAT solver, decreases i to the highest bit index j that is smaller than the current i such that $s_j = 1$ and sets η to $\sum_{j=i}^{k} 2^j \times s_j$, that is, sets η to the number represented by the sequence $s_k, ..., s_i$ assuming that all bits with index smaller than i are set to 0.

Example 3.1.5. Consider once again the partial MaxSAT instance from example 3.1.2, (ϕ_H, ϕ_S) where $\phi_S = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2)\}$.

$|\phi_S| = 3$, so 2 bits are needed for this instance. The algorithm starts by assigning the value 1 to the bit with index 1 and 0 to the other. Therefore, η is initialized with value 2. The SAT solver is invoked with

[1] This is similar to the computation of the middle value in algorithm 2, where η is the middle value.

the following propositional formula:

$$\phi_W \cup [r_1 + r_2 + r_3 \leq 2]_{CNF} \tag{3.7}$$

Suppose that the SAT solver returns the assignment $\nu = \{(x_1, 0), (x_2, 0), (r_1, 1), (r_2, 0), (r_3, 0)\}$. The propositional formula is satisfiable, therefore the algorithm must compute the sequence of bits s_1, s_0. The number of relaxation variables that were assigned the truth value 1 in ν is 1 (only r_1), so $s_1 = 0$ and $s_0 = 1$. The new value for i is 0 and for η is $2^1 \times s_1 + 2^0 \times s_0 = 1$.

Now the SAT solver is invoked with the following propositional formula:

$$\phi_W \cup [r_1 + r_2 + r_3 \leq 1]_{CNF} \tag{3.8}$$

The new propositional formula is satisfiable as well, but this time, since $i = 0$, there exists no bit in the sequence s_1, s_0 with index smaller than i, so i is assigned -1 and the algorithm terminates. The new *assignment* returned by the SAT solver is the solution returned by the algorithm.

3.1.3 Core Guided Algorithms

Core Guided algorithms require the SAT solver to return an unsatisfiable core if the given set of clauses is unsatisfiable. The information given by the unsatisfiable core is used to relax soft clauses on demand.

The first Core Guided algorithm, proposed by Fu and Malik [2006], is presented in algorithm 4. This algorithm allows for soft clauses to be relaxed more than once. As a result, the same soft clause may have more than one relaxation variable assigned to it. The initial working formula ϕ_W is composed of the union of all the clauses in ϕ_H and ϕ_S, and then the SAT solver is invoked iteratively on ϕ_W until it returns that ϕ_W is satisfiable. Each time the SAT solver returns that ϕ_W is unsatisfiable, the algorithm relaxes each soft clause in the unsatisfiable core ϕ_C provided by the SAT solver with a fresh relaxation variable. The previous version of that soft clause in ϕ_W is replaced by the new relaxed version. Additionally, an at-most-1 constraint over the new relaxation variables is added to ϕ_W. The optimum solution is the assignment returned by the satisfying invocation of the SAT solver (figure 3.2).

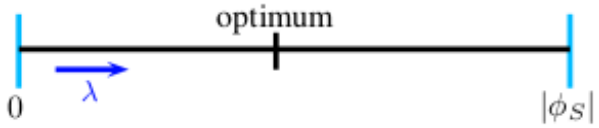

Figure 3.2: Core guided search.

The function *SAT*(ϕ) in algorithm 4 now returns an unsatisfiable core ϕ_C if ϕ is unsatisfiable.

Example 3.1.6. Consider the partial MaxSAT instance from example 3.1.2, (ϕ_H, ϕ_S) where $\phi_S = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2)\}$.

The initial working formula is:

$$\phi_W = \phi_H \cup \phi_S \tag{3.9}$$

Algorithm 4: *Core Guided Unsat-Sat* algorithm for partial MaxSAT (Fu and Malik [2006])

Input: (ϕ_H, ϕ_S)
$\phi_W \leftarrow \phi_H \cup \phi_S$
$R \leftarrow \emptyset$
$status \leftarrow UNSAT$
$\nu \leftarrow nil$
while $status = UNSAT$ **do**
  $(status, \nu, \phi_C) \leftarrow$ `SAT(`ϕ_W`)`
  if $status = UNSAT$ **then**
    $R_{core} \leftarrow \emptyset$
    foreach *soft clause* $c \in \phi_C$ **do**
      $(c_r, R_c) \leftarrow$ `Relax(`$\{c\}$`)`
      $\phi_W \leftarrow (\phi_W \backslash \{c\}) \cup c_r$
      $R_{core} \leftarrow R_{core} \cup R_c$
    end
    $\phi_W \leftarrow \phi_W \cup [\sum_{r \in R_{core}} r \leq 1]_{CNF}$
    $R \leftarrow R \cup R_{core}$
  end
end
$\nu \leftarrow$ `RemoveVariables(`ν, R`)`
return ν

The SAT solver returns that ϕ_W is unsatisfiable. Suppose that the SAT solver also returns the following unsatisfiable core:

$$\phi_C = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2)\} \tag{3.10}$$

The new working formula solved by the SAT solver is:

$$\phi_W = \phi_H \cup \{(x_1 \vee x_2 \vee r_1), (\neg x_1 \vee r_2), (\neg x_2 \vee r_3)\} \cup [r_1 + r_2 + r_3 \leq 1]_{CNF} \tag{3.11}$$

The new working formula is satisfiable, so the algorithm terminates and the optimum solution is the assignment returned by the SAT solver.

Variants of algorithm 4 for solving weighted partial MaxSAT are presented in the literature (Ansótegui et al. [2009a], Manquinho et al. [2009]). The difference is in the relaxation of the soft clauses. The weighted algorithms first compute the minimum weight w_{min} among the soft clauses in ϕ_C. Then, the soft clauses in ϕ_C are replicated and added to the working formula. All those replicas are relaxed with fresh relaxation variables and are assigned weight w_{min}. The weight of all the original soft clauses in ϕ_C is decremented by w_{min}. Note that the original soft clauses in ϕ_C with weight w_{min} will have a weight of 0 after this process. In the end of the relaxation, this means that the original soft clause was replaced by the relaxed replica.

Example 3.1.7. Let $\phi_C = \{(x_1 \vee x_2, 5), (\neg x_1, 4), (\neg x_2, 6)\}$ be an unsatisfiable core, returned by a SAT solver, composed only of weighted soft clauses, and ϕ_W the corresponding working formula. The minimum weight among the soft clauses in ϕ_C is $w_{min} = 4$. After the relaxation process, ϕ_W has the soft

Algorithm 5: *Core Guided Unsat-Sat* algorithm for partial MaxSAT that only adds at most 1 relaxation variable per soft clause (Marques-Silva and Planes [2007])

Input: (ϕ_H, ϕ_S)
1 $\phi_W \leftarrow \phi_H \cup \phi_S$
2 $status \leftarrow UNSAT$
3 $\nu \leftarrow nil$
4 $\lambda \leftarrow 0$
5 $R \leftarrow \emptyset$
6 **while** $status = UNSAT$ **do**
7 $\quad (status, \nu, \phi_C) \leftarrow \texttt{SAT}(\phi_W \cup [\sum_{r \in R} r \leq \lambda]_{CNF})$
8 $\quad$ **if** $status = UNSAT$ **then**
9 $\quad\quad$ **foreach** *soft clause* $c \in \phi_C$ *that is not relaxed* **do**
10 $\quad\quad\quad (c_r, R_c) \leftarrow \texttt{Relax}(\{c\})$
11 $\quad\quad\quad \phi_W \leftarrow (\phi_W \backslash \{c\}) \cup c_r$
12 $\quad\quad\quad R \leftarrow R \cup R_c$
13 $\quad\quad$ **end**
14 $\quad\quad \lambda \leftarrow \lambda + 1$
15 $\quad$ **end**
16 **end**
17 $\nu \leftarrow \texttt{RemoveVariables}(\nu, R)$
18 **return** ν

clauses in ϕ_C replaced by the following set of clauses:

$$\{(x_1 \vee x_2, 1), (\neg x_2, 2), (x_1 \vee x_2 \vee r_1, 4), (\neg x_1 \vee r_2, 4), (\neg x_2 \vee r_3, 4)\} \tag{3.12}$$

Marques-Silva and Manquinho [2008] proposed another variant of algorithm 4 that reduces the number of auxiliary variables in the working formula. This variant does not use relaxation variables to relax the soft clauses. Instead, the auxiliary variables of the bitwise encoding (Frisch et al. [2005], Prestwich [2007]) of the at-most-1 constraint are used. Essentially, the soft clauses in ϕ_C are directly assigned a bit string as described in appendix A, and this way the level of indirection introduced by the relaxation variables is removed.

The next algorithm, presented in algorithm 5, was proposed by Marques-Silva and Planes [2007] and also reduces the number of auxiliary variables in the working formula by only adding at most 1 relaxation variable per soft clause and by maintaining only 1 at-most-k constraint instead of adding a new at-most-1 constraint at each iteration. Algorithm 5 is somehow similar to algorithm 4, but now the cycle (line 9) that relaxes the soft clauses in ϕ_C checks if each soft clause $c \in \phi_C$ has already been relaxed, and only if c has not been relaxed yet then c is relaxed with a fresh relaxation variable. Also, a lower bound λ and a set R of the current relaxation variables in ϕ_W are maintained. λ is incremented at each iteration and R is updated each time a soft clause is relaxed.

Example 3.1.8. Consider the partial MaxSAT instance, (ϕ_H, ϕ_S) where $\phi_S = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2), (x_3), (\neg x_3)\}$.

The initial values for λ and R is 0 and $\emptyset$ respectively. Therefore, the initial working formula is:

$$\phi_W = \phi_H \cup \phi_S \tag{3.13}$$

The SAT solver returns that ϕ_W is unsatisfiable. Suppose that the SAT solver also returns the following unsatisfiable core:

$$\phi_C = \{(x_1 \vee x_2), (\neg x_1), (\neg x_2)\} \tag{3.14}$$

The clauses in ϕ_C are all soft and have not been relaxed yet. The new relaxation variables r_1, r_2 and r_3 are added to R, and λ is incremented to 1.

The new working formula solved by the SAT solver is:

$$\begin{aligned}\phi_W =& \phi_H \cup \\ & \{(x_1 \vee x_2 \vee r_1), (\neg x_1 \vee r_2), (\neg x_2 \vee r_3), (x_3), (\neg x_3)\} \cup \\ & [r_1 + r_2 + r_3 \leq 1]_{CNF}\end{aligned} \tag{3.15}$$

The new working formula is also unsatisfiable and the following unsatisfiable core is returned:

$$\phi_C^{'} = \{(x_3), (\neg x_3)\} \tag{3.16}$$

Since the clauses in $\phi_C^{'}$ are also all soft and have not been relaxed yet, R is updated with the new relaxation variables r_4 and r_5. λ is incremented to 2.

Finally, the SAT solver is invoked with the following working formula:

$$\begin{aligned}\phi_W =& \phi_H \cup \\ & \{(x_1 \vee x_2 \vee r_1), (\neg x_1 \vee r_2), (\neg x_2 \vee r_3), (x_3 \vee r_4), (\neg x_3 \vee r_5)\} \cup \\ & [r_1 + r_2 + r_3 + r_4 + r_5 \leq 2]_{CNF}\end{aligned} \tag{3.17}$$

This last working formula is satisfiable, so the algorithm terminates and the optimum solution is the assignment returned by the SAT solver on this last invocation.

A variant of algorithm 5 was proposed by Marques-Silva and Planes [2008] that also maintains an upper bound μ, that is updated like the upper bounds in algorithms 1 and 2. In this variant, the value k in the at-most-k constraint of the working formula is set to $\mu - 1$ instead of λ. The main cycle stops when the the condition $\mu > \lambda + 1$ does not hold anymore or when the SAT solver returns an unsatisfiable core ϕ_C such that all the soft clauses in ϕ_C have already been relaxed. The solution is the last assignment returned by the SAT solver.

Another variant of algorithm 5 was proposed by Ansótegui et al. [2009a]. This variant extends the working formula with additional at-least-k constraints to further guide the SAT solver. All the soft clauses are relaxed in the beginning, like the algorithms of sections 3.1.1 and 3.1.2, and every unsatisfiable core returned by the SAT solver is stored in a set C at each iteration. After adding a new unsatisfiable core ϕ_C to C, the algorithm counts how many cores in C are contained in ϕ_C[2]. Then, a new at-least-k constraint is added on the relaxation variables of the soft clauses in ϕ_C, where the k value is the number

[2]Throughout the entirety of this document is assumed that set operations over unsatisfiable cores only consider the soft clauses in those cores. Also note that, because of this assumption, we can identify an unsatisfiable core through the relaxation variables of the soft clauses it contains.

of unsatisfiable cores in C that are contained in ϕ_C.

An extension of this last variant was proposed by Ansótegui et al. [2009b] that uses several at-most-k constraints instead of just 1. These at-most-k constraints are based on what is referred to as a *cover* of C. An unsatisfiable core ϕ_C is a *cover* of C if, for each $\phi_C^{'} \in C$ such that $\phi_C \cap \phi_C^{'} \neq \emptyset$, then $\phi_C^{'} \subseteq \phi_C$. Instead of using a single at-most-k constraint on all the relaxation variables in the working formula, this algorithm computes at each iteration the set of covers of C and a new set of at-most-k constraints, one per cover of C. The constrained variables in an at-most-k constraint related to a given cover Γ are the relaxation variables in Γ, and the k value for that constraint is the number of unsatisfiable cores in C that are contained in Γ. Note that when a new set of at-most-k constraints is computed, the old set is replaced. An extension of this algorithm, based on sets of covers, for solving weighted partial MaxSAT that does not require clause replication can be found in the literature (Ansótegui et al. [2010]).

The third algorithm, proposed by Heras et al. [2011], is almost identical to algorithm 2. The only difference is that soft clauses are relaxed on demand, like in algorithm 5, instead of relaxing all of them at the beginning. A variant was proposed by Heras et al. [2011] as well that exploits disjoint unsatisfiable cores and uses several at-most-k constraints instead of just one, similarly to the algorithm proposed by Ansótegui et al. [2009b]. The concept of disjoint cores is equivalent to the concept of covers described before. A set C of disjoint cores is maintained by the algorithm. A lower bound λ_i, an upper bound μ_i and a middle value ρ_i are maintained for each core $\phi_i \in C$. When invoking the SAT solver, an at-most-k constraint $\sum_{r \in R_i} r \leq \rho_i$, where R_i is the set of relaxation variables in ϕ_i, is encoded in the working formula for each unsatisfiable core $\phi_i \in C$. If the SAT solver returns satisfiable, the upper bounds μ_i of each core in C are refined. If the SAT solver returns unsatisfiable, the algorithm checks if the set of soft clauses of the new unsatisfiable core ϕ_C intersects any of the sets of soft clauses of the cores in C. If so, the intersecting cores are merged into a single core[3], and the old intersecting cores are replaced in C by the new merged core. If, on the other hand, ϕ_C is disjoint, then this new core is added to C. The algorithm terminates when the condition $\lambda_i + 1 < \mu_i$ does not hold for all the cores $\phi_i \in C$.

An evaluation of all the algorithms described so far, since the start of section 3.1.1, can be found in the literature (Morgado et al. [2013]).

3.1.4 Branch and Bound Algorithms

This section presents a family of algorithms for MaxSAT that does not rely on iteratively invoking a SAT solver. Instead, these algorithms are variants of a basic Branch and Bound procedure, presented in algorithm 6. The differences are mostly in how the lower bounds are estimated and how the variables are selected at each step.

For this algorithm we need to introduce the following definitions and notation.

Definition 3.1.1. An assignment $\nu : X \rightarrow \{0, 1\}$ is a *complete assignment* if $\nu(x)$ is defined for all the variables $x \in X$. An assignment is a *partial assignment* if it is not complete.

[3]The lower bound and upper bound for the merged core is the sum of the lower bounds and upper bounds of the intersecting cores respectively.

Definition 3.1.2. An *inference rule* is a procedure that generates new clauses or variable assignments from a known subset of clauses.

Definition 3.1.3. *Unit propagation* is an inference rule that generates a new variable assignment from a unit clause.

Notation 3.1.1. Let ϕ be a set of clauses and x a variable in ϕ. We denote as ϕ_x the set of clauses obtained from replacing x in ϕ with 1 and simplifying ϕ afterwards. We denote as $\phi_{\neg x}$ the set of clauses obtained from replacing x in ϕ with 0 and simplifying ϕ afterwards.

It is assumed that the process of simplifying a set of clauses ϕ includes, at least, the following two steps: delete clauses with literals that are assigned the truth value 1; delete literals that are assigned the truth value 0. Inference rules can be used to further transform a propositional formula ϕ into a simpler equivalent propositional formula and to speed up Branch and Bound algorithms. Note that inference rules applied to a MaxSAT instance ϕ must preserve the number of unsatisfied clauses in ϕ. Some simple inference rules for MaxSAT are:

- *pure literal rule* (Niedermeier and Rossmanith [2000]): if literals containing a given variable x are all positive (negative), then x is assigned the truth value 1 (0);
- *dominating unit clause rule* (Niedermeier and Rossmanith [2000]): if the number of clauses that contain x ($\neg x$) is not greater than the number of unit clauses $\neg x$ (x), then x is assigned the truth value 0 (1);
- *complementary unit clause rule* (Niedermeier and Rossmanith [2000]): a pair of unit clauses of the form $\{(x), (\neg x)\}$ is replaced by an empty clause;
- *almost common clause rule* (Bansal and Raman [1999]): a pair of clauses of the form $\{(x \vee l_1 \vee ... \vee l_n), (\neg x \vee l_1 \vee ... \vee l_n)\}$ is replaced by the clause $\{(l_1 \vee ... \vee l_n)\}$.

Some more complex inference rules are presented in the literature (Larrosa and Heras [2005], Heras and Larrosa [2006]).

The Branch and Bound algorithm starts by computing an estimation of the upper bound μ on the number of clauses unsatisfied by an optimum assignment. One trivial possibility for the initial μ is the number of clauses in ϕ. Another possibility is to run a local search solver on ϕ for a short period of time and choose the best value it returns as the initial μ.

Afterwards, a recursive procedure begins that computes the optimum assignment for a given set of clauses ϕ. This procedure receives as input the current formula ϕ, the partial assignment ν that originated ϕ and the current upper bound μ[4]. It starts by checking if there are any variables in ϕ still left to be assigned. If not, then all clauses in ϕ have either been satisfied or unsatisfied, and the unsatisfied clauses are the empty clauses in ϕ. If there are variables still left to assign, then the procedure computes a lower bound λ on the number of clauses in ϕ that will be unsatisfied if the current partial assignment ν is extended to a complete assignment. The simplest method to compute λ is to determine the number

[4]For the *BranchAndBoundMaxSAT* procedure, μ is the cost of the best solution found so far.

Algorithm 6: Basic *Branch and Bound* algorithm for MaxSAT (Alsinet et al. [2003], Li et al. [2007], Lin et al. [2008], Borchers and Furman [1999])

```
Input: φ
(ν, μ) ← BranchAndBoundMaxSAT(φ, ∅, ComputeInitialμ(φ))
return ν

Procedure BranchAndBoundMaxSAT(φ, ν, μ)
    if φ = ∅ or φ has only empty clauses then
        return (ν, NumberOfEmptyClauses(φ))
    end
    λ ← NumberOfEmptyClauses(φ) + Underestimation(φ)
    if λ ≥ μ then
        return (ν, μ)
    end
    x ← SelectVariable(φ)
    (ν1, μ1) ← BranchAndBoundMaxSAT(φx, ν ∪ (x, 1), μ)
    (ν0, μ0) ← BranchAndBoundMaxSAT(φ¬x, ν ∪ (x, 0), Min(μ, μ1))
    μ ← Min(μ, μ1, μ0)
    if μ = μ1 then
        ν ← ν1
    else if μ = μ0 then
        ν ← ν0
    end
    return (ν, μ)
```

of clauses unsatisfied by the current ν (Borchers and Furman [1999]). If μ is not higher than λ, then an optimum solution can not be found by extending ν, so the procedure returns. Otherwise, an unassigned variable x in ϕ is selected heuristically and the optimum assignments for ϕ_x and $\phi_{\neg x}$ are computed recursively. The procedure then checks if any of the optimum values returned for ϕ_x and $\phi_{\neg x}$ is better than the current μ. If so, the minimum of those values and the corresponding assignment are returned.

The functions in the algorithm work as follows:

- *ComputeInitial*$\mu(\phi)$ computes an initial upper bound on the number of clauses unsatisfied by an optimum solution;
- *NumberOfEmptyClauses*(ϕ) returns the number of empty clauses in ϕ;
- *Underestimation*(ϕ) computes a lower bound on the number of non-empty clauses in ϕ that will become unsatisfied if all variables in ϕ are assigned. The most basic method computes the number of disjoint conflicting pairs of unit clauses in ϕ (Wallace and Freuder [1996]). Other methods either search ϕ for disjoint inconsistent subsets of clauses of the form $\{(l_1), ..., (l_n), (\neg l_1 \vee ... \vee \neg l_n)\}$ (Alsinet et al. [2004], Shen and Zhang [2004]), apply unit propagation to ϕ to detect disjoint inconsistent subsets of clauses (Li et al. [2005, 2006], Darras et al. [2007], Lin et al. [2008]) or reduce ϕ to an instance of another problem, such as Integer Linear Programming, solve its relaxation instance and choose the corresponding solution as a lower bound (Xing and Zhang [2005], Pipatsrisawat et al. [2008], Ramírez and Geffner [2007]);
- *SelectVariable*(ϕ) heuristically selects a variable in ϕ to be assigned next. Various MaxSAT solvers apply variants of the Jeroslow-Wang rule (Jeroslow and Wang [1990]) for SAT that prioritize variables that occur often in binary clauses.

Algorithm 6 only solves instances of MaxSAT. Branch and Bound algorithms for weighted MaxSAT are presented in the literature (Alsinet et al. [2005, 2008], Larrosa et al. [2008], Lin and Su [2007], Xing and Zhang [2004, 2005], Borchers and Furman [1999]). In these algorithms, μ and λ are bounds on the sum of the weights of the clauses unsatisfied by an optimum assignment. The main differences are in the functions *ComputeInitial*$\mu(\phi)$, *NumberOfEmptyClauses*(ϕ) and *Underestimation*(ϕ). For the weighted case, *ComputeInitial*$\mu(\phi)$ and *Underestimation*(ϕ) compute bounds on the sum of weights of unsatisfied clauses, and the function *NumberOfEmptyClauses*(ϕ) is replaced by a function that returns the sum of the weights of the empty clauses in ϕ.

Algorithms for weighted partial MaxSAT can be found in the literature (Heras et al. [2008], Pipatsrisawat et al. [2008], Ramírez and Geffner [2007]). These algorithms take as input an additional set of hard clauses ϕ_H and, at each step, verify if one of the clauses in ϕ_H is unsatisfied by the current partial assignment ν. If so, then no complete assignment obtained from extending ν exists that is a valid solution, and the recursive procedure returns[5]. Also, unit propagation (and possibly other inference rules) is applied on ϕ_H. If a new variable assignment is inferred from ϕ_H, then that assignment is propagated to the soft clauses as well.

3.2 Parallel MaxSAT

In this section, parallel algorithms for MaxSAT are reviewed. First, algorithms based on a Porfolio approach are presented. Next, algorithms that split the Search Space across the available processors are described. Finally, some aspects of clause sharing among processes are detailed.

3.2.1 Portfolio Approach

A parallel solver based on a Portfolio approach executes various sequential solvers in parallel with different configurations. It relies on the fact that specific algorithms can solve a subset of problem instances much faster than most of the other algorithms, while being slower than other algorithms in other instances.

The most basic parallel Portfolio algorithm for MaxSAT is composed only of two threads. One of the threads searches on the upper bound value with a Linear Search algorithm (section 3.1.1) and the other thread executes a Core Guided algorithm (section 3.1.3) to search on the lower bound. Such an algorithm was implemented in PWBO (Martins et al. [2011b], Martins [2013], Martins et al. [2012b]) for solving weighted partial MaxSAT. However, these two threads do not solve the problem instance independently. The two threads cooperate by sharing learned clauses among them. Clause sharing among threads will be detailed in section 3.2.3. Both threads terminate when at least one of them has determined an optimum solution.

Parallel algorithms with 4 and 8 threads are also proposed for PWBO (Martins et al. [2011a], Martins [2013], Martins et al. [2012b]) that can solve partial MaxSAT. These algorithms are very similar to the

[5]The procedure *BranchAndBoundMaxSAT* could return a value of ∞ so that the previous recursive step would not choose the invalid partial assignment when computing the minimum among upper bounds.

algorithm with just two threads. Half of the threads execute a Linear Search algorithm and the other half execute a Core Guided algorithm. To achieve more diversity in the search process, the threads executing the Linear Search algorithm use different encodings for the at-most-k constraints (appendix A), and the threads executing the Core Guided algorithm use different encodings for the at-most-1 constraints. The threads share learned clauses among them, like in the algorithm with two threads.

However, while a Linear Search thread t_i is solving its set of clauses, it repeatedly checks if any of the other Linear Search threads t_j has determined a better upper bound μ_j. If so, then thread t_i interrupts its execution, sets $\mu_i = \mu_j$ and updates its at-most-k constraint with the new μ_i. Next, thread t_i starts a new iteration of the Linear Search algorithm. Also, if a Linear Search thread t_i determines a new tighter upper bound μ_i and is unaware of any upper bounds from other threads better than μ_i, it exports μ_i to all the other Linear Search threads.

Similarly, whenever a Core Guided thread computes a new unsatisfiable core, it exports that core to all other Core Guided threads. A Core Guided thread t_i repeatedly checks if any of the other threads has computed a new unsatisfiable core. If so, then thread t_i imports the new unsatisfiable core and incorporates it into its current propositional formula, relaxing its soft clauses and adding a new at-most-1 constraint.

All the threads terminate when at least one of them has determined an optimum solution.

3.2.2 Search Space Splitting

The Search Space Splitting approach aims to divide the search space in subspaces and assign each subspace to a different thread. Ideally, these subspaces should be disjoint. Another algorithm proposed for PWBO (Martins [2013], Martins et al. [2012b]) applies this method and considers the search space to be the range of possible values for the number of unsatisfied clauses in the optimum solution.

Given n threads, one thread executes a Core Guided algorithm (section 3.1.3) to search on the lower bound, initialized as 0, and another thread executes a Linear Search algorithm (section 3.1.1) to search on the upper bound, initialized as the number of soft clauses plus 1. The remaining $n-2$ threads search on a *local upper bound value*. Supposing that ϕ_S is the set of soft clauses in the problem instance, thread t_1 is the Core Guided thread and thread t_n is the Linear Search thread, the local upper bound μ_i of thread t_i, $1 < i < n$, is initialized with the expression $(i-1) \times \lfloor \frac{|\phi_S|+1}{n-1} \rfloor$. Threads $t_2, ..., t_{n-1}$ are referred to as the *Local Linear Search* threads. Note that the subspaces obtained from this approach are not necessarily disjoint. All threads may share learned clauses among them like in the Portfolio approach.

The Local Linear Search algorithm is very similar to the Linear Search algorithm. The main difference is that the k value of the at-most-k constraint of a thread t_i is the local upper bound μ_i, limiting t_i's search to assignments that unsatisfy no more than μ_i soft clauses. This at-most-k constraint is classified as a *thread bound constraint*. If the SAT solver returns that the working formula is unsatisfiable, then the lower bound is updated accordingly; otherwise, the upper bound is updated. Updates to the lower and upper bounds are exported to the other threads. At this point, t_i has to compute a new local upper bound μ_i and start a new search on that value. During its execution, t_i also has to repeatedly check if any of

the other threads has exported a lower or upper bound that excludes μ_i from the current possibilities for optimum solutions. This happens if the lower bound becomes higher than μ_i or if the upper bound becomes smaller than μ_i. In this case, t_i has to abort its execution, compute a new local upper bound and restart. Note that the Linear Search thread also has to abort its execution and restart if a better upper bound is exported by a Local Linear Search thread.

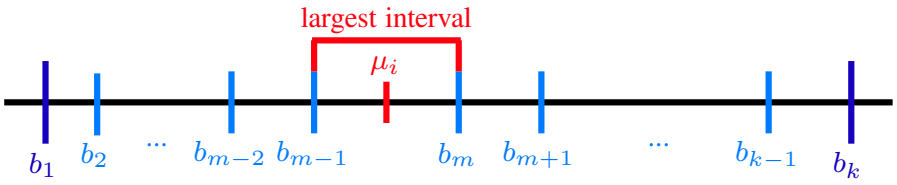

Figure 3.3: New local bound computation.

The computation of a new local upper bound is done as follows. Let t_i be an aborted Local Linear Search thread and $B = \{b_1, ..., b_k\}$ an ordered set where b_1 is the lower bound, b_k is the upper bound and all the other b_j are the local upper bounds currently being searched by the active Local Linear Search threads. t_i chooses a pair (b_{m-1}, b_m) of contiguous values in B such that $b_m - b_{m-1} \geq b_j - b_{j-1}$ for all $1 < j \leq k$. The new μ_i is given by the expression $\frac{b_m + b_{m-1}}{2}$ (figure 3.3). Finally, μ_i is added to B.

In the literature (Martins [2013], Martins et al. [2012b]), the previous Search Space Splitting approach is evaluated experimentally and compared with the Portfolio approach described in the previous section.

Another Search Space Splitting approach has been proposed for parallel SAT solving that uses guiding paths (Zhang et al. [1996]). Given n[6] threads, $log_2\, n$ distinct variables of the problem instance are selected. With $log_2\, n$ variables we have n possible partial assignments. Each of these partial assignments is a guiding path, and each guiding path encodes a subspace. A different guiding path is assigned to each thread. Then, each thread simplifies its copy of the set of clauses of the problem instance according to its guiding path, and solves the resulting set of clauses. Note that a thread t_i may determine that no satisfying assignment exists in its subspace faster than the remaining threads. In this case, a new guiding path must be computed and assigned to t_i. Böhm and Speckenmeyer [1996] proposed a dynamic work stealing procedure to compute these new guiding paths and balance the workload among threads.

3.2.3 Clause Sharing

In the case of parallel MaxSAT solvers, there exist a few restrictions with respect to which clauses may be shared among which threads. It is also needed to have some care when importing clauses shared by other threads.

In the clause learning process, an implication graph is used to express the implication relationships among the variable assignments. In the implication graph, vertexes represent variable assignments and edges represent clauses that lead to the corresponding implication. The new learned clauses are a product of the analysis of that graph.

[6]For simplicity, we assume that n is a power of 2.

The following definitions are needed for the rest of this section.

Definition 3.2.1. A *soft learned clause* is a clause inferred from an implication graph that uses at least one soft clause. A *hard learned clause* is a clause inferred from an implication graph that uses only hard clauses.

Definition 3.2.2. A thread bound constraint is a *local constraint*. A learned clause c is a *local constraint* if the implication graph, used to infer c, uses at least one *local constraint*.

Note that the clauses used to encode a thread bound constraint are, by definition, local constraints.

State-of-the-art SAT solvers are inspired by the DPLL (Davis et al. [1962]) algorithm. This algorithm corresponds to a depth-first search with backtracking, where at each branching step a variable is selected to be assigned.

Definition 3.2.3. A *decision level* of an assigned variable is the depth of the search tree at which the variable was assigned.

Learned clauses can be shared among threads as long as the following conditions are met (Martins [2013]):

- A hard learned clause c from a Core Guided thread can be shared with all other threads if c does not include auxiliary variables used in the cardinality constraint encodings;
- A hard learned clause c from a (Local) Linear Search thread can be shared with all other threads if c is not a local constraint and does not include auxiliary variables used in the cardinality constraint encodings;
- A hard learned clause c, that is also a local constraint, from a (Local) Linear Search thread t_i can be shared with another thread t_j if c does not include auxiliary variables used in the cardinality constraint encodings and the upper bound μ_j of t_j is equal or smaller than μ_i of t_i.

Whenever a thread checks for lower or upper bound updates from other threads, it also imports learned clauses shared by the other threads. However, some care is needed when incorporating these clauses into the current search context[7] to ensure correctness of the SAT solver (Martins [2013]). Let c be a clause being imported by a thread:

- If c is a unit clause, then the search restarts and the corresponding variable is assigned accordingly;
- If c is unit in the current context, then the SAT solver backtracks to the highest decision level of the assigned variables in c, assigns the unassigned variable accordingly, applies unit propagation and then the search is resumed;
- If c is unsatisfied in the current context, then the SAT solver backtracks to the highest decision level of the variables in c and then applies conflict analysis to allow further backtracking;

[7] The current search context can be denoted by the partial assignment composed of the variable assignments that lead to the current node in the search tree, including the variable assignments derived by inference.

- If exactly one literal in c is satisfied in the current context, all other literals in c are unsatisfied and the decision level of the satisfied literal is higher than the decision level of all the unsatisfied literals, then the SAT solver backtracks to the highest decision level of the variables in the unsatisfied literals;
- In all other cases, c can be added immediately.

Sharing learned clauses is expected to improve the performance of a parallel algorithm, but may lead to an exponential memory blow up and some of the shared clauses may not become useful[8] at any point throughout the search process. Therefore, not all learned clauses should be shared among threads, and some heuristic should be used to choose which clauses to share or import. There are three types of heuristics: static, dynamic and freezing.

The static heuristics only allow a learned clause c to be shared if it is within a given cutoff. These heuristics may take into account the number of literals in c (Martins et al. [2012b], Hamadi et al. [2009a]). In this case, c is shared only if it has a number of literals smaller than a given constant, because smaller clauses are expected to be more useful than large ones. Another static heuristic considers how many decision levels are there among the variables in c. This measure is referred to as the *literal block distance* (Audemard and Simon [2009]). This second heuristic only allows c to be shared if its literal block distance is below a given cutoff.

However, the size of learned clauses tends to increase over time. Therefore, applying heuristics that rely on a static cutoff may lead the parallel algorithm to stop sharing learned clauses at a certain point in time. Dynamic heuristics allow for the cutoff to be dynamically improved throughout the execution of the algorithm. A dynamic heuristic was proposed and implemented in the parallel SAT solver ManySAT (Hamadi et al. [2009b]) that maintains a cutoff per pair of threads. Each cutoff is periodically improved based on the throughput and on a quality measure of the learned clauses being shared between the two threads.

The static and dynamic heuristics dictate if a clause should be exported or not. However, freezing heuristics (Martins et al. [2012a]) are used to decide if the incorporation of a shared clause should be delayed or not. Freezing heuristics allow a clause to be imported only if that clause is expected to be useful in the near future. If not, then that clause is stored in a set of frozen clauses. A frozen clause is reevaluated later to determine if it has become likely to be useful. If a frozen clause is evaluated more than a given constant number of times, then that clause is deleted.

An experimental evaluation and comparison of the aforementioned clause sharing heuristics can be found in the literature (Martins et al. [2012a], Martins [2013]).

[8] A learned clause is useful if it is unsatisfied or unit in the current context.

Chapter 4

Distributed MaxSAT

This section presents the two distributed algorithms that were studied and implemented. The first algorithm is an adaptation of the parallel Search Space Splitting approach, described in section 3.2.2, for a distributed context. The second algorithm is based on the guiding paths splitting strategy (Zhang et al. [1996], Heule et al. [2012]), which was shown to improve the performance of sequential SAT solvers and to be successful in parallel and distributed SAT algorithms.

In both algorithms there are two main types of processes: a single mediator process and multiple slave processes. The mediator process is responsible for splitting the problem instance into tasks and assigning those tasks to the slave processes. The mediator also handles all the communication between processes: slave processes send and receive messages to and from the mediator, but no messages are exchanged among slave processes. The mediator is also responsible for detecting when an optimum solution has been found.

The slave processes' main purpose is to wait for a task from the mediator, process the received task and send the result back to the mediator. Tasks are processed with the aid of a SAT solver. The SAT solver's code was incorporated directly into the distributed solver's code to avoid the need of intermediary files, allowing more efficient and easier manipulation and invocation of the SAT solver.

Both algorithms were implemented in C++ and MPI was used for inter-process communication. The chosen SAT solver was Glucose 2.3 (Audemard et al. [2013]). The CNF encoding used for the Cardinality Constraints was an optimized version of the Sorters encoding described in appendix A, the same encoding used in OpenWBO (Martins et al. [2014b]).

4.1 Search Space Splitting Algorithm

In the distributed Search Space Splitting algorithm the instance is divided in the same way as in the parallel version described in section 3.2.2. Given n processes, the algorithm is composed by 1 mediator, 1 core guided process, 1 linear search process and $n - 3$ local linear search processes.

The mediator process is depicted in figure 4.1. Before assigning tasks to the remaining processes, the mediator initializes a sorted set that will store, through the execution of the algorithm, the best lower

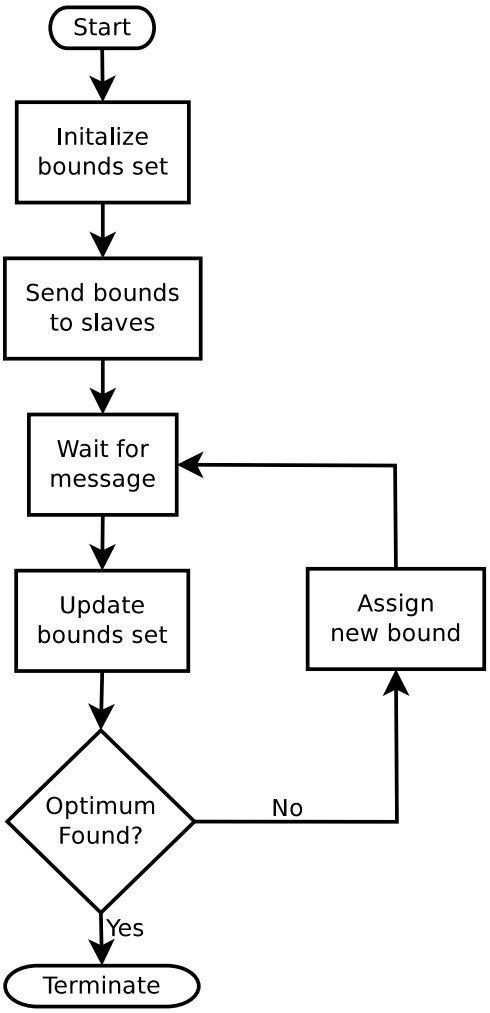

Figure 4.1: Search Space Splitting mediator process.

and upper bounds determined so far and the local bounds currently being searched by each local linear search process. The mediator starts by computing an initial upper bound by invoking the SAT solver on the CNF formula ϕ_H. On an earlier version of the algorithm, the mediator used to choose $|\phi_S| + 1$ as the initial upper bound, but it was observed experimentally that computing the initial upper bound with the aid of the SAT solver can greatly reduce the size of the initial search space, consequently reducing the amount of memory needed to encode the cardinality constraints and the amount of time needed to determine the optimum. It was also observed that formula ϕ_H is usually solved quickly by the SAT solver.

If ϕ_H is not satisfiable, then the algorithm terminates immediately returning that the instance is not satisfiable. If ϕ_H is satisfiable, then the mediator computes the number of soft clauses satisfied by the model returned by the SAT solver, chooses that value as the initial upper bound and adds it to the set. The initial lower bound is 0 and the initial local bounds are computed as follows: given k local linear search processes, $p_1, ..., p_k$, and an initial upper bound μ, the initial local bound assigned to process p_i is given by $i \times \lfloor \frac{\mu}{k+1} \rfloor$. After adding these bounds to the bounds set, each of them is sent to the corresponding local linear search process, and $\mu - 1$ is sent to the linear search process.

Example 4.1.1. Let $\mu = 37$ be an initial upper bound. Given 5 local linear search processes, the initial

bounds set is $B = \{0, 6, 12, 18, 24, 30, 37\}$.

Next, the mediator enters a loop, waiting for upper and lower bounds from the slave processes and assigning new bounds to them until the optimum has been found. If the mediator receives a lower bound λ from a process p, all values smaller than λ are removed from the bounds set, and λ is added to the set. Otherwise, if the mediator receives an upper bound μ, all values larger than μ are removed and μ is added to the set. Next, a new bound is computed and sent to process p. No message is sent to the slave processes solving bounds smaller than λ or bigger than μ. Doing so would imply changing the code of the SAT solver in order to check for new messages from the mediator process and end the search prematurely. This feature was not implemented in the algorithm because we aimed to develop a distributed solver completely independent of the SAT solver. This way, one can choose to use one among multiple SAT solvers, as long as it implements the interface used by the distributed solver. However, it would be interesting to evaluate if this small feature would have a significant impact in the performance of the distributed solver, but this was not done due to time constraints.

Let $B = \{b_0, b_1, ..., b_k, b_{k+1}\}$ be the current bounds set. If p is the core guided search process, $b_0 + 1$ is sent to p. If p is the linear search process, $b_{k+1} - 1$ is sent to p instead. If p is a local linear search process, then the mediator chooses a pair (b_{m-1}, b_m) of contiguous values such that $b_m - b_{m-1} \geq b_j - b_{j-1}$ for all $1 < j \leq k$. A new local bound b is computed with the expression $\frac{b_m + b_{m-1}}{2}$. b is then added to B and sent to process p.

Example 4.1.2. Let $B = \{5, 12, 22, 27, 40\}$ be a bounds set. Suppose that a message from p_1 is received stating that 26 is an upper bound. The new bounds set will be $B_1 = \{5, 12, 17, 22, 26\}$ where 17 is p_1's new local bound. Suppose that, afterwards, a message from p_2 is received stating that 19 is a lower bound. The new bounds sets will be $B_2 = \{19, 22, 24, 26\}$ where 24 is p_2's new local bound.

Figure 4.2 depicts the behavior of the linear search and local linear search processes. These processes start by relaxing all the soft clauses with new relaxation variables. Then, each process enters an infinite loop, waiting for a bound b from the mediator and computing if b is an upper bound or a lower bound. For the linear search process, b will always be the current upper bound minus 1, but for the local linear search processes, b is a local bound chosen by the mediator as described above.

Let ϕ_H be the set of hard clauses, ϕ_S^r the set of relaxed soft clauses and R the set of relaxation variables. A linear search process p determines if b is an upper or a lower bound by invoking the SAT solver on the CNF formula $\phi_H \cup \phi_S^r \cup [\sum_{r \in R} r \leq b]_{CNF}$. After solving b, the mediator responds with a new bound to be solved $b^{'}$. If b was determined to be a lower bound, $b^{'}$ will be larger than b. On the other hand, if b was determined to be an upper bound, $b^{'}$ will be smaller than b.

If $b^{'} < b$, then it is possible to obtain an equivalent CNF encoding for $\sum_{r \in R} r \leq b^{'}$ from $[\sum_{r \in R} r \leq b]_{CNF}$ without removing clauses from the latter. This can be achieved merely by assigning truth values to certain auxiliary variables of the CNF encoding for $\sum_{r \in R} r \leq b$. Therefore, we only need to add the corresponding unit clauses to $[\sum_{r \in R} r \leq b]_{CNF}$ to obtain $[\sum_{r \in R} r \leq b^{'}]_{CNF}$. Martins et al. [2014a] take advantage of this property to improve the performance of linear search algorithms by avoiding the need to rebuild the SAT solver. This way, all clauses learned so far by the SAT solver are maintained between

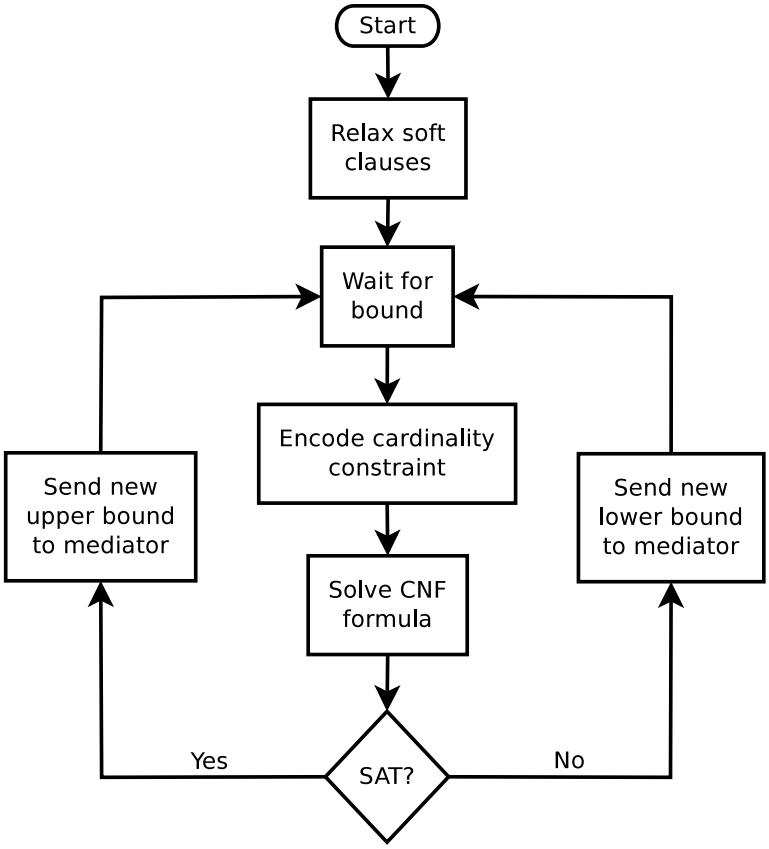

Figure 4.2: Search Space Splitting linear search process.

invocations.

However, the property above is not true if $b^{'} > b$. One could remove the clauses of $[\sum_{r \in R} r \leq b]_{CNF}$ from the SAT solver and then encode $\sum_{r \in R} r \leq b^{'}$, but this may invalidate a subset of the learned clauses stored in the SAT solver. Therefore, in the local linear search processes, when b is an upper bound the SAT solver is maintained and the new cardinality constraint encoding is obtained from the previous one, but when b is a lower bound the SAT solver is rebuilt.

If the SAT solver returns that the CNF formula is satisfiable, then the process retrieves a model ν from the solver and computes the number of soft clauses μ unsatisfied by ν. Next a message is sent to the mediator notifying that μ is an upper bound. On the other hand, if the CNF formula is unsatisfiable, then another message is sent instead notifying the mediator that b is a lower bound.

The behavior of the core guided process is depicted in figure 4.3. That behavior is very similar to algorithm 5, the main difference being the additional exchange of messages with the mediator. The core guided process starts by adding relaxation variables to the soft clauses. Then it enters a cycle of invocations to the SAT solver with unsatisfiable CNF formulas until a satisfiable formula is found.

The cycle starts by invoking the SAT solver with the complements of the relaxation variables as assumptions. The assumptions are used to force the soft clauses to be satisfied. If it is not possible, then the algorithm retrieves a conflicting subset of the assumptions and relaxes the corresponding soft clauses.

The algorithm also maintains the best lower bound computed locally and the best lower bound that

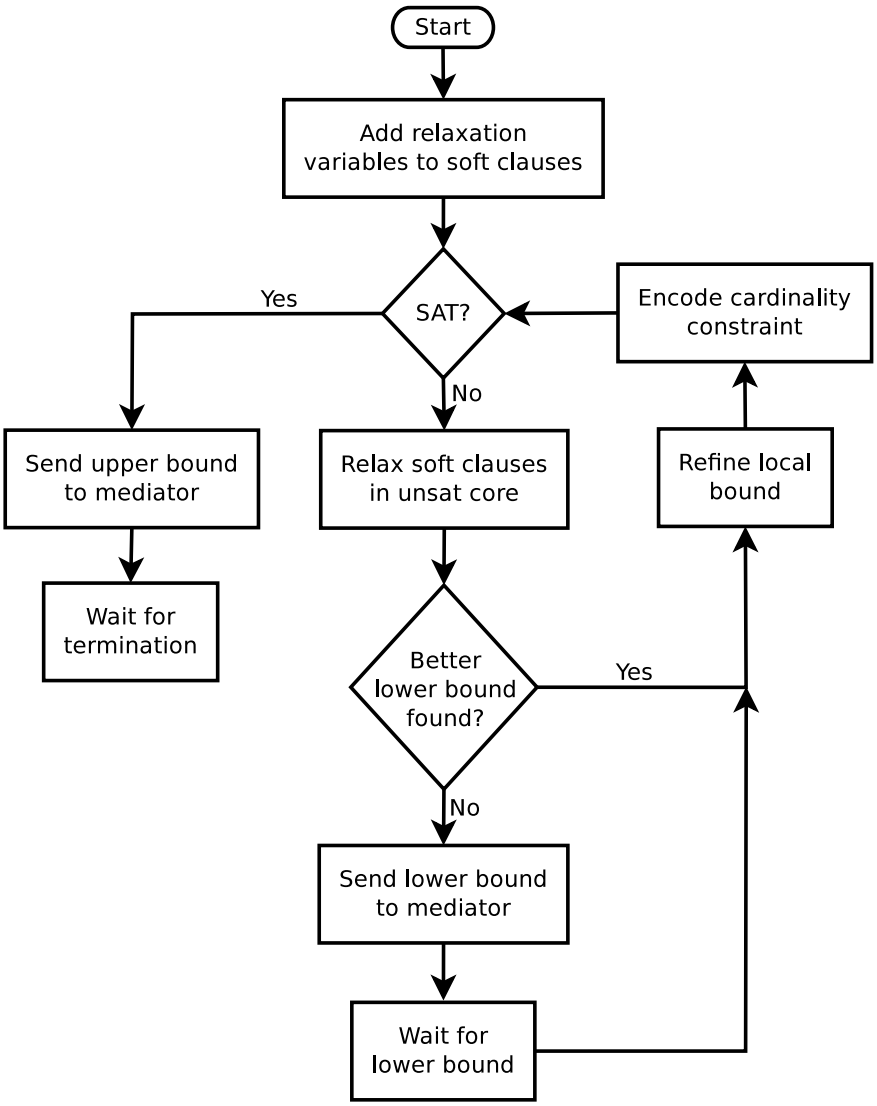

Figure 4.3: Search Space Splitting core guided process.

was received from the mediator so far. The local lower bound is initialized as 0. If the local lower bound is better than the lower bound received from the mediator, then the local lower bound is sent to the mediator, and the core guided process waits for a new lower bound from the mediator. The local lower bound is then refined, and a cardinality constraint is encoded into the CNF formula over the current set of relaxation variables and with the new local lower bound as right hand side. For unweighted MaxSAT instances, the local lower bound is refined merely by an increment operation.

Finally, the cycle restarts by invoking the SAT solver again, but the assumptions corresponding to the soft clauses that were relaxed are excluded. The cycle continues until either the SAT solver returns that the CNF formula is satisfiable or until the process is terminated by the mediator. In the first case, the core guided process sends the local bound to the mediator stating that it is an upper bound and awaits termination.

Example 4.1.3. Let ϕ_W be the current working formula and $\Lambda = \{\neg r_1, \neg r_2, \neg r_3, \neg r_4\}$ the current set of assumptions. Suppose that the SAT solver, after being invoked on ϕ_W under Λ, returns the conflict

$\{\neg r_1, \neg r_4\}$. The new set of assumptions will be $\Lambda^{'} = \{\neg r_2, \neg r_3\}$.

An optimum is found when the lower bound λ and the upper bound μ stored in the mediator meet the condition $\lambda + 1 = \mu$. When this happens, the mediator aborts the execution of the remaining processes and terminates, returning μ as the optimum. If requested, slave processes may store the model returned by the SAT solver on the last satisfiable invocation. In this case, when μ is determined to be the optimum, the mediator requests the corresponding model from the slave process that sent the first message received by the mediator stating that μ is an upper bound. In an unweighted MaxSAT instance, this model will be an assignment that unsatisfies exactly μ soft clauses. The model is returned by the mediator together with the optimum.

It may be the case that a point in the execution of the algorithm is reached where there are more local linear search processes than local bounds to be solved. Given N processes, this happens when $\mu - \lambda < N - 2$. When this situation is detected by the mediator, the algorithm starts a portfolio mode of execution. When a local linear search process requests a new bound, a local bound is chosen randomly among the local bounds still in the bounds set at that moment. The mediator sends the chosen local bound to the local linear search process and notifies it that the mode of execution should swap to portfolio mode.

When a local linear search process enters a portfolio mode, it randomizes the polarities of the variables in the SAT solver before each SAT call. The polarity of a variable x dictates which truth value for x will be tested first in the execution of the SAT algorithm. If x has positive polarity, then the value 1 is tested first. If x has negative polarity instead, then 0 is tested first. This randomizes the search process of the SAT solver in each local linear search process, usually inducing different processes to search the same bound differently.

Two versions of this algorithm were implemented and tested. In one version, a local linear search process p rebuilds the SAT solver whenever it determines that a given local bound b is a lower bound. This is because the new local bound $b^{'}$ that will be assigned to p is guaranteed to be larger than b. Therefore, adding unit clauses to the SAT solver does not suffice to update the cardinality constraint from b to $b^{'}$, and to encode the new cardinality constraint the SAT solver must also be rebuilt entirely.

The second version makes use of assumptions and of the fact that the mediator computes an initial upper bound to only build the SAT solver once and never have to rebuild it again. This second version is referred to as the Fully Incremental Search Space Splitting algorithm.

Suppose that process p has an upper bound μ encoded in its current cardinality constraint. The mediator assigns a new local bound b to p. Instead of adding a set U of unit clauses to the SAT solver, since p does not know if b is an upper or a lower bound, p invokes the SAT solver with U as the set of assumptions. This updates the cardinality constraint without altering the SAT instance stored in the solver. If b is an upper bound, then p adds unit clauses to its SAT solver to update the cardinality constraint from μ to $b - 1$. If b is a lower bound, the SAT solver remains unchanged. It has been observed that always invoking the same SAT solver incrementally boosts the performance of sequential MaxSAT algorithms (Martins et al. [2014a]).

4.2 Guiding Paths with Lookahead Algorithm

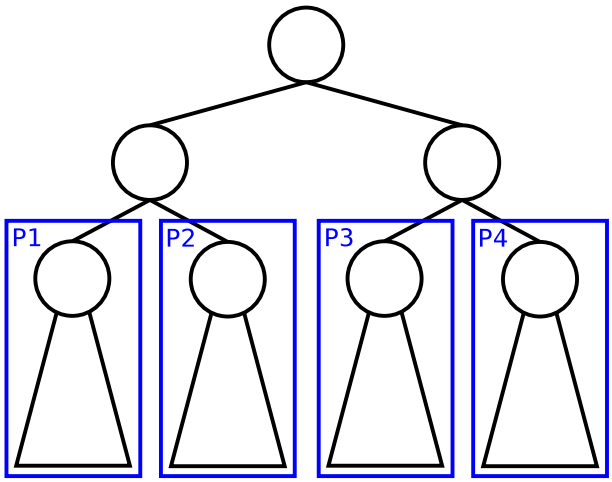

Figure 4.4: Example of a search tree split among 4 processes.

The distributed Guiding Paths with Lookahead algorithm is based on the guiding paths approach described for parallel SAT solving in section 3.2.2. A distributed algorithm for SAT based on guiding paths has already been proposed and evaluated with PaMiraXT (Schubert et al. [2009]). Consider the search space of a SAT instance as a binary tree. Each tree node corresponds to a variable and each of its edges corresponds to an assignment to that variable (0 or 1). A path from the root node to a leaf node corresponds to a complete assignment. All possible assignments are encoded in that tree. The SAT problem can be interpreted as, given a SAT instance, determine if a path exists in its search tree from the root to a leaf node that corresponds to a satisfying assignment. What the guiding paths approach does is to split the search tree into sub-trees and assign each of them to a distinct process (figure 4.4). In partial MaxSAT, the goal is to find a path from the root to a leaf node corresponding to an assignment that satisfies all hard clauses and minimizes the number of unsatisfied soft clauses.

Heule et al. [2012] have proposed recently an improved parallel SAT algorithm based on guiding paths that initially uses a *lookahead* solver to split the search tree in what they refer to as *cubes*. Each cube corresponds to a guiding path. Lookahead solvers apply sophisticated reasoning at each branching step in order to guide the search more effectively. These solvers perform well in random instances, but fail to be competitive in industrial instances. However, the experimental analysis performed by Heule et al. [2012] shows that using lookahead solvers to split the tree into guiding paths and then solving each guiding path with a CDCL solver (Marques-Silva et al. [2009]) can further boost the performance of the SAT algorithm. The algorithm described throughout the rest of this section is an extension of the previous approach to distributed MaxSAT. Given n processes, the following algorithm is composed by 1 mediator and $n-1$ guiding path solver processes.

The behavior of the mediator process is depicted in figure 4.5. Initially, the mediator computes and stores guiding paths until an initial upper bound is received from one of the slave processes. This initial upper bound is computed in the same way as the initial upper bound computed by the mediator in the Search Space Splitting algorithm (section 4.1). However, in the Guiding Paths algorithm one of the slave

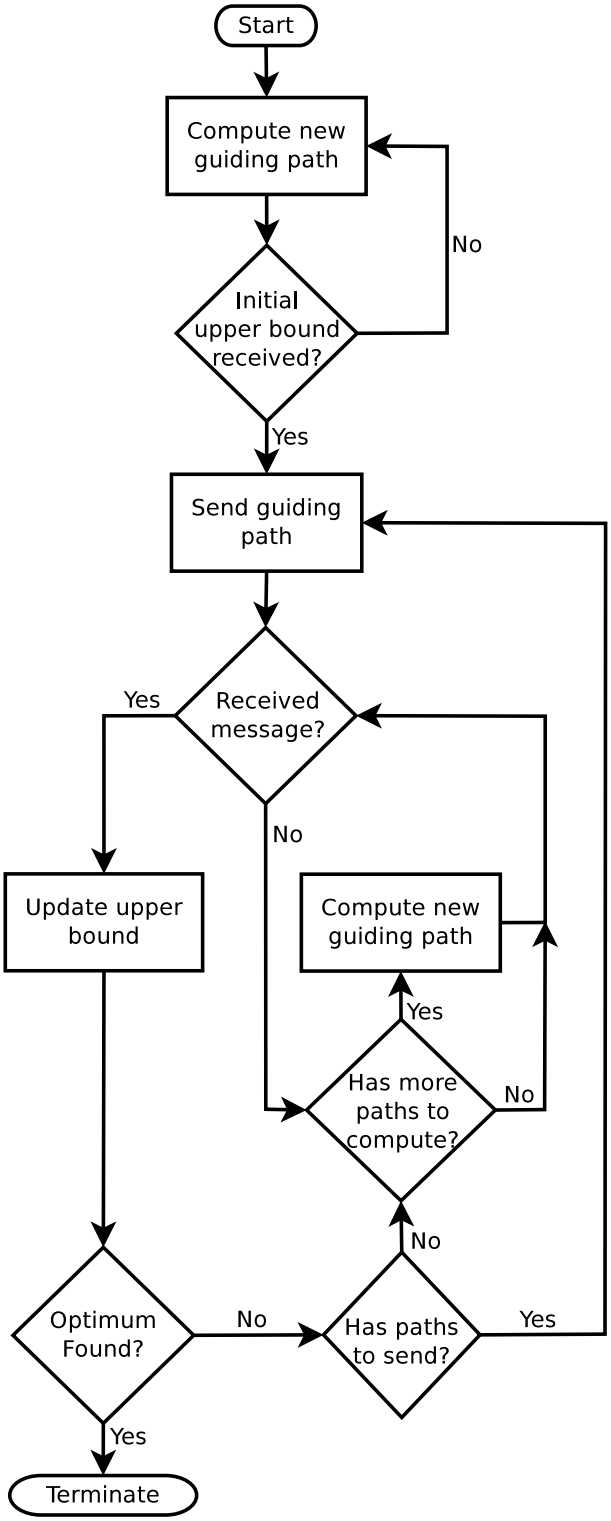

Figure 4.5: Guiding Paths with Lookahead mediator process.

processes is responsible for computing the initial upper bound, allowing the mediator to start computing guiding paths in parallel. When the initial upper bound is received, the mediator starts fulfilling the requests from the slave processes, sending them guiding paths to solve coupled with the best upper bound computed so far.

A guiding path is represented as a set of literals, where a positive literal x_i corresponds to an as-

Algorithm 7: Guiding Path generation algorithm (Heule et al. [2012])

```
1  Procedure GenerateGuidingPaths(φ, C, D, I, θ)
2      IncrementCutoff(θ)
3      (φ, I) ← Propagate(φ, D, I)
4      if φ is unsatisfied by D ∪ I or |D| + log2 |φ| > 25 then
5          DecrementCutoff(θ)
6      end
7      if φ is unsatisfied by D ∪ I then
8          AnalyzeAndLearn(φ, D, I)
9          return C
10     end
11     if |D| × |D ∪ I| > θ × | Vars(φ) | then
12         return C ∪ {D}
13     end
14     x ← ChooseVariable(φ, D, I)
15     l ← ChoosePolarity(φ, x)
16     C ← GenerateGuidingPaths(φ, C, D ∪ {l}, I, θ)
17     return GenerateGuidingPaths(φ, C, D ∪ {¬l}, I, θ)
```

signment of 1 to x_i and a negative literal $\neg x_i$ corresponds to an assignment of 0 to x_i. In other words, a guiding path is a set of literals that must be satisfied throughout the process of solving that path.

Example 4.2.1. Let $g = \{x_1, \neg x_2, \neg x_3, x_4, \neg x_5\}$ be a guiding path. g translates into the partial assignment $\nu = \{(x_1, 1), (x_2, 0), (x_3, 0), (x_4, 1), (x_5, 0)\}$.

While waiting for new messages from the slave processes, the mediator continues computing and storing new guiding paths in a set C. When there is a pending message from a slave process p, the mediator first checks if $|C| = 0$. If so, then the mediator first needs to compute a new guiding path g and add it to C. Next, the mediator chooses a path $g \in C$ and sends it to p. Note that the mediator also sends the best upper bound μ computed so far together with g to process p.

After solving g, process p may respond either with a new upper bound $\mu^{'}$, smaller than μ, or with a subset g_c of the literals in g, referred to as a conflict. The literals in g_c cannot be simultaneously satisfied by an assignment that unsatisfies less than μ soft clauses. If the solver process responds with an upper bound μ', the mediator checks if μ' is smaller than the best upper bound computed so far (stored in the mediator). If so, the upper bound stored in the mediator is replaced by μ'.

The pseudo-code for the guiding path generation procedure is presented in algorithm 7. This procedure receives as input a CNF formula ϕ, the set C of guiding paths computed so far by the procedure, the current partial assignment D, the set I of literals that are implied by the partial assignment D and a cutoff value θ. In practice, the guiding path generation is implemented as an object with a method that computes one guiding path at a time, in the same order as they would be added to set C in algorithm 7. However, the guiding path generation procedure is presented as a recursive algorithm for the sake of presentation. Formula ϕ used for guiding path generation in the MaxSAT algorithm is the set of hard clauses. It was observed experimentally that considering just the set of hard clauses in the guiding path generation procedure provided better results than considering both the hard clauses and the soft clauses. The functions in the procedure work as follows:

- *IncrementCutoff*(θ) increases the cutoff value θ destructively[1]. In practice, θ is incremented by 5%, like in the literature (Heule et al. [2012]);
- *DecrementCutoff*(θ) decreases the cutoff value θ destructively. In practice, θ is decremented by 30%, like in the literature (Heule et al. [2012]);
- *Propagate*(ϕ, D, I) applies unit propagation to formula ϕ. New literals are added to set I that represent assignments implied by the partial assignment D on ϕ;
- *AnalyzeAndLearn*(ϕ, D, I) applies conflict analysis (Marques-Silva and Sakallah [1996], Zhang et al. [2001]) to learn a new clause;
- *Vars*(ϕ) returns the set of all variables in ϕ;
- *ChooseVariable*(ϕ, D, I) chooses a new variable heuristically to be added to the partial assignment D. The addition of a new variable assignment to D is referred to as a *decision*. A rank is assigned to each variable not in $D \cup I$ and the one with the highest rank is chosen;
- *ChoosePolarity*(ϕ, x) chooses heuristically which truth value will be assigned first to variable x. Returns either x or $\neg x$.

The algorithm starts by incrementing the cutoff value θ (line 2). This is done to prevent θ from being reduced too much by the decrement rule in line 4 and generating too small guiding paths. The initial cutoff value is 1000 as specified by Heule et al. [2012]. Next, unit propagation is applied to simplify ϕ and update set I (line 3). The algorithm then checks if ϕ is unsatisfied by the current partial assignment D and implied assignments in I (line 4). In this case, θ is decremented (line 5). θ is also decremented if $|D| + \log_2 |\phi| > 25$ (line 4). This rule takes into account the number of decisions and the size of the CNF formula in order to prevent the guiding path generation process of going too deep in the search tree and generating too many guiding paths that are also too large. Heule et al. [2012] propose, for SAT, a value of 30 for the right hand side of the condition in line 4, but we observed experimentally that a value of 25 provided better results for MaxSAT.

If ϕ is unsatisfied, the procedure applies conflict analysis (Marques-Silva and Sakallah [1996], Zhang et al. [2001]) and learns a new clause (line 8), like in a CDCL SAT solver (Marques-Silva et al. [2009]). This may prevent the procedure from generating guiding paths that unsatisfy ϕ. If ϕ is not unsatisfied the algorithm checks if the cutoff has been triggered (line 11). If so, the current partial assignment D is returned as a guiding path. The cutoff condition takes into account the number of decisions and the total number of assignments, explicit and implied, in the current node of the search tree.

If the cutoff is not triggered, then an unassigned variable x is chosen heuristically to be added to D (line 14). Another heuristic is used to decide which truth value will be tested first (line 15). Algorithm 7 is then repeated for x and $\neg x$ (lines 16 and 17). Heule et al. [2012] propose two different heuristics for ranking a variable, and both require an heuristic value to be computed for x and $\neg x$, denoted as $eval(x)$ and $eval(\neg x)$. The rank of a variable x is $eval(x) \times eval(\neg x)$ and ties are broken by $eval(x) + eval(\neg x)$.

[1] Destructive modifications to a given parameter are globally visible. Therefore, destructive modifications are propagated upwards in the call chain of the procedure.

Given a variable x, $eval_{var}(x)$ denotes the number of variables that are assigned by unit propagation after the assignment $x = 1$. $eval_{var}(\neg x)$ is the same, but for $x = 0$. The second heuristic, denoted as $eval_{cls}$, weights clauses depending on their length. $eval_{cls}(x)$ ($eval_{cls}(\neg x)$) is the sum of the weights of the clauses that are reduced[2] by the assignment $x = 1$ ($x = 0$) but are not satisfied. The clauses are weighted in a way such that a clause with length k has a weight five[3] times larger than a clause with length $k + 1$. Heule et al. [2012] observed that, for SAT solving, $eval_{cls}$ is effective in solving instances that have none or few binary clauses, while $eval_{var}$ is more effective on industrial instances.

Example 4.2.2. Let $\phi = \{(x_1 \vee x_2 \vee x_3), (x_2 \vee \neg x_3), (\neg x_1 \vee x_2)\}$ be a CNF formula. In ϕ, $eval_{var}(\neg x_2) = 2$ and $eval_{cls}(\neg x_3) = 6$ (clauses with length 2 and 3 have weights 5 and 1 respectively if 3 is the maximum clause size).

A variation of the $eval_{cls}$ heuristic, which will be referred to as $eval_{wl}$, was implemented and tested. The only difference is, given a literal l, instead of considering all the clauses in ϕ, only the clauses "watching" (Moskewicz et al. [2001]) literal l are considered in the computation of $eval_{wl}(l)$. The clauses "watching" l are a subset of the clauses that contain l. In state-of-the-art SAT solvers, each clause "watches" only two of its literals at each instant. Suppose that a given clause c is watching literals l_1 and l_2 and that l_1 and l_2 are neither satisfied or unsatisfied. If l_1 becomes unsatisfied, then another non-unsatisfied literal $l_3 \in c$ is chosen to be watched instead of l_1. If l_3 is satisfied, then c is satisfied. If no such literal like l_3 exists, then the clause has become unit and the variable in l_2 is assigned accordingly. If the SAT algorithm backtracks, the watched literals remain unchanged. Watched literals are a lazy data structure that greatly improve the performance of SAT solvers. It was observed experimentally that the $eval_{wl}$ heuristic was more effective than the $eval_{var}$ and $eval_{cls}$ heuristics in this distributed MaxSAT algorithm.

After choosing a variable x, now a truth value must be chosen to be tested first. For SAT, Heule et al. [2012] aim to improve the performance on satisfiable instances. Therefore, they choose to explore first the branch $x = 1$ if $eval(x) < eval(\neg x)$. However, in MaxSAT all guiding paths will inevitably result in one unsatisfiable call to the SAT solver. Besides, MaxSAT is an optimization problem, not a decision problem. We choose the direction based on the number of clauses that will be unsatisfied after assigning x. The branch to be explored first is the one that unsatisfies a smaller number of soft clauses. Ties are broken choosing the direction that satisfies more soft clauses. The rationale for this is that the branch that unsatisfies less soft clauses is more likely to reach an upper bound closer to the optimum, and since upper bounds are shared among slave processes, this may help skip unnecessary satisfiable calls to the SAT solver.

However, the polarity heuristic is only significant at the beginning of the search, when the mediator is generating guiding paths, because the mediator sorts guiding paths as they are generated. Therefore, in the long run, the sorting criterion dominates the polarity heuristic. Initially, the criterion used to sort the guiding paths was very similar to the polarity heuristic, sorting guiding paths by number of unsatisfied soft clauses. But we observed experimentally that sorting guiding paths by the total of assigned variables

[2] A clause is reduced whenever one of its literals is assigned value 0.
[3] Same value used for SAT (Heule et al. [2012]).

(decision and implied), when the path is returned, lead to better results. The priority is given to the least restricting guiding path and ties are broken by choosing the one that was generated first.

Example 4.2.3. Suppose that algorithm 7 returns the guiding paths g_1, g_2, g_3 and g_4, exactly in this order. The dimensions of D when each path was returned were 10, 8, 11 and 7 respectively, and the dimensions of I were 100, 92, 97 and 103. The priority values for each of the paths are $p(g_1) = 10 + 100 = 110$, $p(g_2) = 8 + 92 = 100$, $p(g_3) = 11 + 97 = 108$ and $p(g_4) = 7 + 103 = 110$. $p(g_1)$ and $p(g_4)$ are the same, but g_1 was returned first. Therefore, the resulting sorted set of guiding paths will be $C = \{g_1, g_4, g_3, g_2\}$.

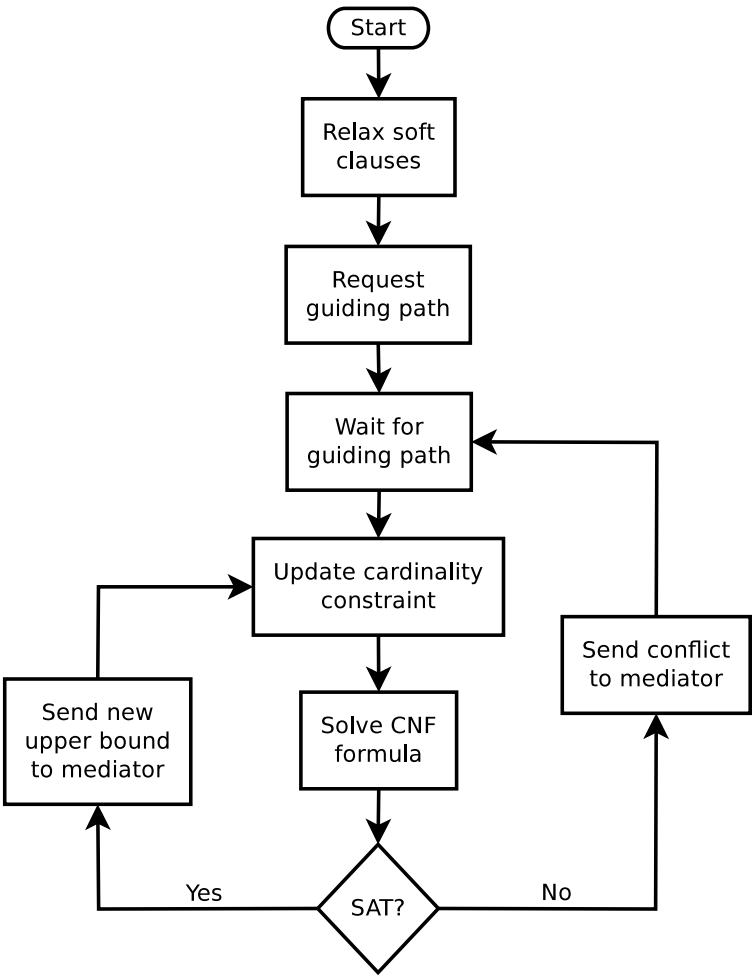

Figure 4.6: Guiding Paths with Lookahead guiding path solver process.

The behavior of the guiding path solver process is depicted in figure 4.6. The computation of the initial upper bound was not included in figure 4.6 for simplicity. These processes apply an adaptation of algorithm 1. A solver process starts by relaxing the soft clauses and requesting a guiding path from the mediator. Next, it enters a loop waiting for guiding paths to solve given by the mediator and applying linear search on those guiding paths.

A guiding path message includes the best upper bound μ found so far. Therefore, when a guiding path is received, the process updates the current cardinality constraint to $\mu - 1$. Having received a guiding path g, the solver process then invokes the SAT solver with g as the set of assumptions. If the SAT solver returns that the formula is satisfiable, the process retrieves a model ν from the SAT solver, computes

its corresponding upper bound μ' and sends μ' to the mediator, like the linear search processes of the algorithm in section 4.1. Afterwards, the process refines the upper bound in the cardinality constraint and continues applying linear search. However, if the SAT solver returned that the formula is unsatisfiable, then a conflicting subset of g is retrieved from the SAT solver and sent to the mediator. The process then waits for a new guiding path from the mediator.

An optimum is found when all guiding paths have been solved or when an empty conflict is obtained. If a given guiding path g results in an empty conflict, then the reason for unsatisfiability does not depend on the assumptions. Therefore, the reason for unsatisfiability is either the cardinality constraint or a conflict in the hard clauses. If there is a conflict in the hard clauses, it is detected in the computation of the initial upper bound. On the other hand, if the reason is the cardinality constraint, then the constraint's right hand side is a lower bound and the optimum was found.

Chapter 5

Experimental Results

In this chapter, both distributed algorithms presented in chapter 4 are evaluated. In sections 5.1 and 5.2, the distributed algorithms are compared with OpenWBO (Martins et al. [2014b]), their sequential counterpart. OpenWBO was configured to use Glucose 2.3 as its SAT solver, the same used in the distributed algorithms. OpenWBO implements algorithms 1 and 4 (see chapter 3). The algorithms are compared in terms of number of solved instances and run time. Also, the algorithms are experimented with different numbers of processes in order to analyze their scalability. The Guiding Paths algorithm is compared with the Search Space Splitting algorithm in section 5.2.

The algorithms were evaluated running on the instances of the industrial and crafted categories of the MaxSAT Evaluation of 2013[1]. Ideally, the algorithms should be run on each instance various times and it only should be considered solved if it was solved on more than half of the runs, but due to time constraints the results throughout this chapter were obtained with just one run per instance. The algorithms were run with a timeout of 1800 seconds (wall clock time) and with a memory limit of $4 \times N$ Gb where N is the number of processes. The execution of the algorithms was monitored using the *runsolver* application (Roussel [2011]). The tests were conducted on 4 machines, each with two Intel Xeon E5-2630V2 processors (2.6GHz, 6C/12T) with 64 Gb of RAM, running Ubuntu 13.10.

5.1 Search Space Splitting Algorithm

Figure 5.1 is a scatter plot comparing the run times on the industrial category of the Search Space Splitting algorithm running with 8 processes on the same machine (SSS-8) with the run times of running the same algorithm with 8 processes, 2 per machine (SSS-2:4). Each point corresponds to a problem instance, where the x-axis is the run time required by SSS-2:4 to solve it and the y-axis is the run time required by SSS-8. Figure 5.2 is the same, but with 16 processes in the same machine (SSS-16) and 4 per machine (SSS-4:4). Instances that are solved trivially by both solvers (in less than 10 seconds) were excluded from the plots.

It is clear that splitting the processes among multiple machines improves the performance of the

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

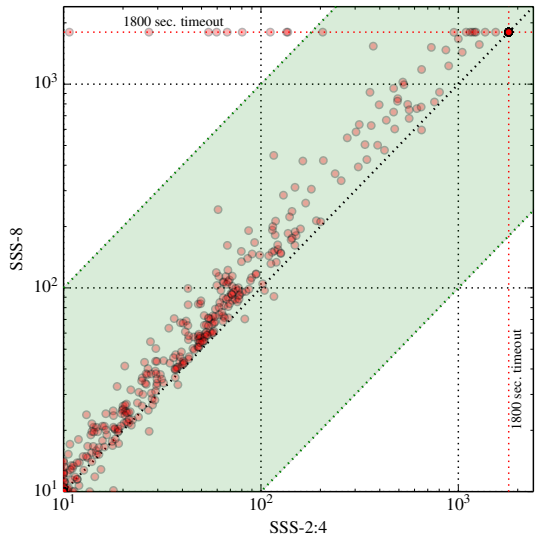

Figure 5.1: Comparison between run times of Search Space Splitting with 8 processes in the same machine (SSS-8) and split among 4 machines (SSS-2:4) for industrial instances.

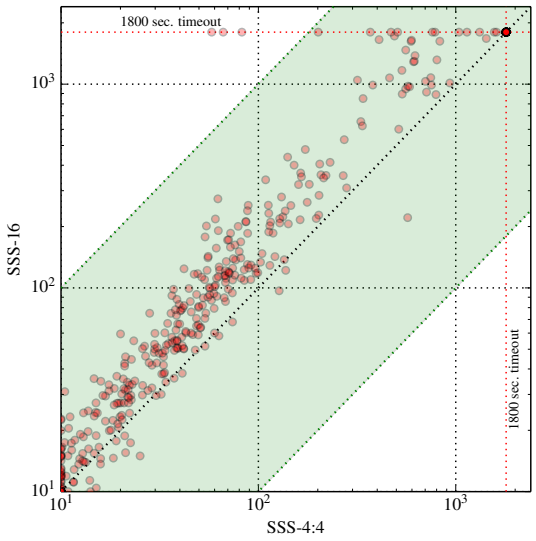

Figure 5.2: Comparison between run times of Search Space Splitting with 16 processes in the same machine (SSS-16) and split among 4 machines (SSS-4:4) for industrial instances.

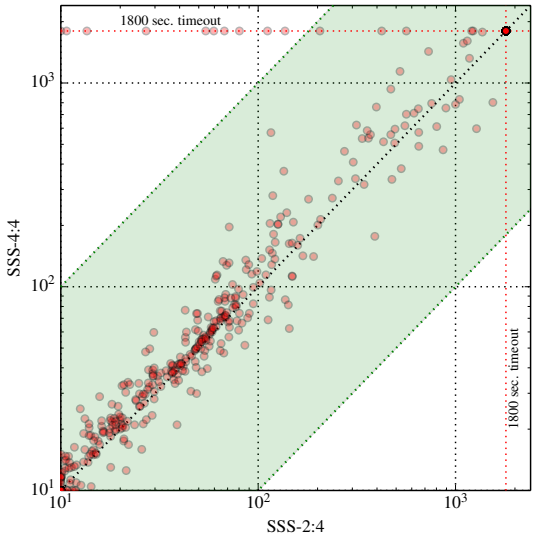

Figure 5.3: Comparison between run times of Search Space Splitting with 8 processes split among 4 machines (SSS-2:4) and 16 split among 4 machines (SSS-4:4) for industrial instances.

algorithm compared to running all processes in the same machine. More 18 instances were solved by SSS-2:4 compared to SSS-8, and more 12 instances were solved by SSS-4:4 than by SSS-16. It was observed that some of these additional instances that were solved were not solved in a single machine due to the memory limit being exceeded. In fact, 10 of the 18 additional instances solved by SSS-2:4 were instances where SSS-8 exceeded the memory limit. However, this happens because *runsolver* (Roussel [2011]) enforces the memory limit only on the processes in the host machine. *runsolver* has no means to monitor the amount of memory that is used by processes in other machines.

Figure 5.3 shows the run times of SSS-2:4 compared with SSS-4:4. SSS-2:4 slightly outperforms SSS-4:4 in terms of run times and SSS-2:4 solves 16 more instances than SSS-4:4, and 14 of those 16

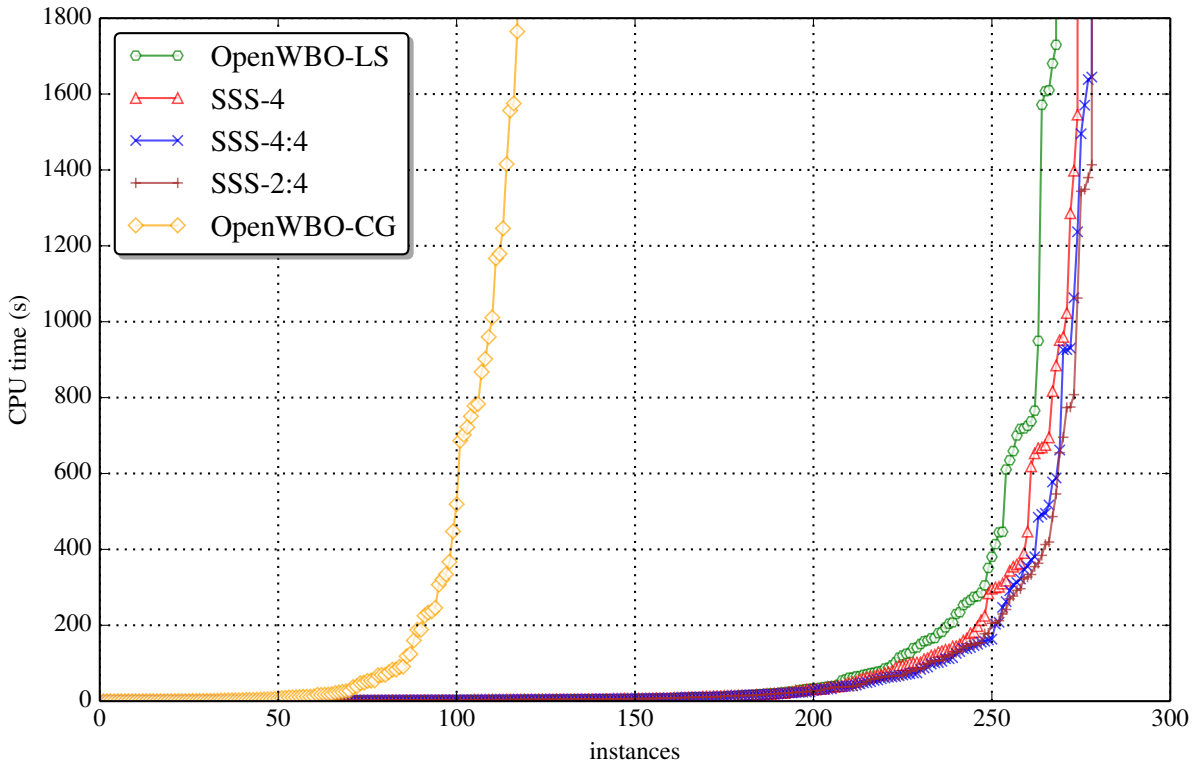

Figure 5.4: Cactus plot with the run times for the instances of the crafted category and the algorithms OpenWBO's core guided and linear search algorithms, and the Search Space Splitting algorithm with 4 processes in the same machine (SSS-4), 8 processes split among 4 machines (SSS-2:4) and 16 processes split among 4 machines (SSS-4:4).

are not solved by SSS-4:4 due to the memory limit being exceeded. In fact, the Search Space Splitting algorithm with 4 processes in the same machine (SSS-4) solves 3 more instances than SSS-4:4. It was observed that there were 4 instances that were solved by SSS-4 but not by SSS-4:4 due to the memory limit being exceeded by SSS-4:4.

This most likely happened because MPI assigned the processes with the largest initial upper bound to the host machine and, as stated above, runsolver only monitors the processes in the host machine. Let R be a set of variables and k_1, k_2 integers such that $k_1 < k_2$. The CNF encoding for $\sum_{r \in R} r \leq k_2$ will be larger than the encoding for $\sum_{r \in R} r \leq k_1$. Since the number of local linear search processes increases, initially there will be more and larger local bounds to be solved with, for example, 16 processes than with 4. This is the main reason why an initial upper bound is computed by the mediator with the aid of the SAT solver instead of choosing $|\phi_S| + 1$ as the initial upper bound, as explained in section 4.1. The larger the right hand side of a cardinality constraint, the more memory will be necessary to encode that constraint into CNF. It was observed experimentally that changing MPI's scheduling policy to assign processes to machines differently caused one of the instances that were not solved by SSS-4:4, due to the memory limit, to become solved.

Figures 5.1, 5.2 and 5.3 show that increasing the number of machines improves the performance of the distributed algorithm, but increasing the number of processes per machine may have a hindering effect. The main reason for this is the contention on the access to primary memory. Computers usually

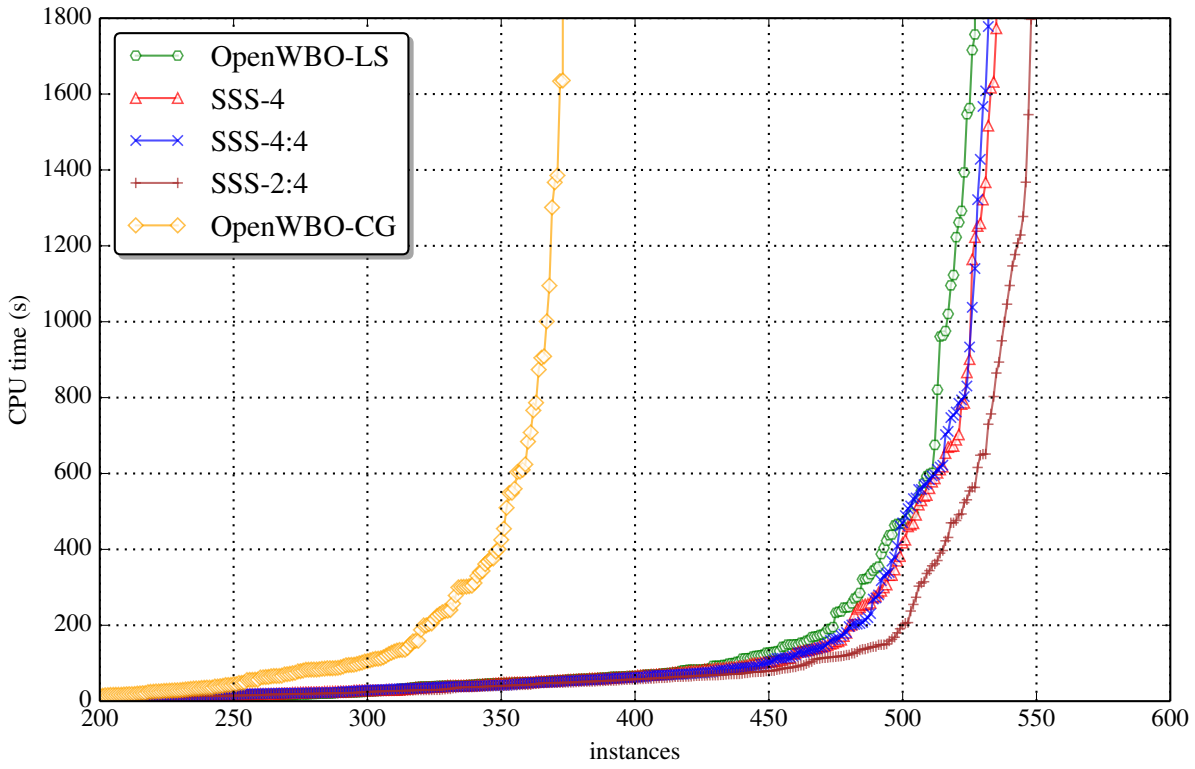

Figure 5.5: Cactus plot with the run times for the instances of the industrial category and the algorithms OpenWBO's core guided and linear search algorithms, and the Search Space Splitting algorithm with 4 processes in the same machine (SSS-4), 8 processes split among 4 machines (SSS-2:4) and 16 processes split among 4 machines (SSS-4:4).

have a single BUS connecting processors to primary memory. Therefore, if one processor is accessing primary memory and another one gets a miss on cache, the second one has to wait for the first to finish before accessing primary memory as well.

Figures 5.4 and 5.5 show cactus plots with the run times of OpenWBO's core guided (OpenWBO-CG) and linear search (OpenWBO-LS) algorithms compared with the run times of SSS-4, SSS-2:4 and SSS-4:4. Each curve represents the results for a single algorithm. Each point on a curve corresponds to a problem instance, where the y-axis is the run time, in seconds, required by the corresponding algorithm to solve that instance. The rightmost curve corresponds to the algorithm that solved the most instances. The curve closer to the bottom corresponds to the algorithm that was the most efficient.

The Search Space Splitting algorithm clearly outperforms OpenWBO in the crafted category. There is a slight increase in performance when executing the distributed algorithm with 8 processes compared to 4, solving 4 more instances. However, there was no gain when increasing the number of processes to 16, solving the same number of instances as SSS-2:4. A similar behavior can be observed in the industrial category, except that the performance of the Search Space Splitting algorithm actually starts degrading with 16 processes. However, most of the instances that were solved by SSS-2:4 and not by SSS-4:4 were due to the memory limit being exceeded by SSS-4:4, as mentioned previously.

In figures 5.6 and 5.7 the run times of the basic and fully incremental versions of the Search Space Splitting algorithm, with 4 and 8 processes respectively, are compared for the crafted category. Figures

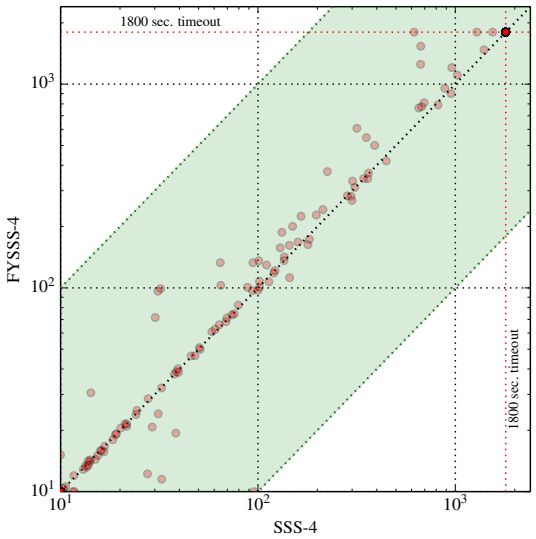

Figure 5.6: Comparison between run times of the basic (SSS-4) and fully incremental (FYSSS-4) versions of the Search Space Splitting with 4 processes, in the same machine, for crafted instances.

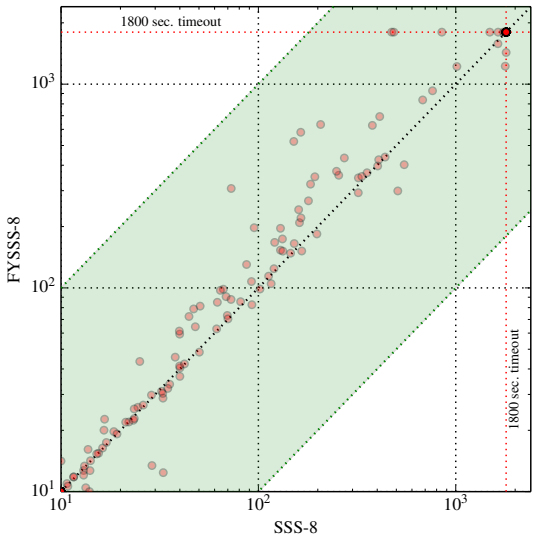

Figure 5.7: Comparison between run times of the basic (SSS-8) and fully incremental (FYSSS-8) versions of the Search Space Splitting with 8 processes, in the same machine, for crafted instances.

5.8 and 5.9 show the same but for the industrial category. We can see that the fully incremental version is competitive with the basic version for industrial instances.

It is known that solving MaxSAT instances incrementally (Martins et al. [2014a]) improves the performance of sequential solvers. CDCL SAT solvers (Marques-Silva et al. [2009]) are known to be very effective in industrial instances due to the conflict driven clause learning mechanism they implement, which makes use of the structure of the instances to acquire knowledge that will prune other parts of the search space. Industrial instances are much more structured than crafted instances. Therefore, CDCL SAT solvers usually are much more effective in the industrial category than in crafted one.

Sequential MaxSAT solvers that are incremental are effective because they maintain the inner state of the SAT solver between calls, reusing knowledge acquired in previous calls. Since CDCL SAT solvers are more effective in industrial instances, it is expected that a MaxSAT solver that uses a CDCL SAT solver will inherit this property, and, as a consequence, incrementality will have much more impact on the performance in industrial instances than in crafted instances, since the re-usage of knowledge has more impact in structured instances.

However, it was expected that the fully incremental version would outperform the basic version at least in the industrial instances. It was observed across a few instances that incrementality improved the individual performance of a local linear search process. But it was observed also that, in the long run, this can lead to the search space being explored differently. For example, in one version a local linear search process may end up solving local bounds that are harder than the bounds that it would solve in the other version. Also, it is expected that, as the number of processes increases, incrementality will have less impact on the performance of the algorithm, since there will be less bounds per local linear search thread to be solved and, consequently, less chances to re-use previous knowledge.

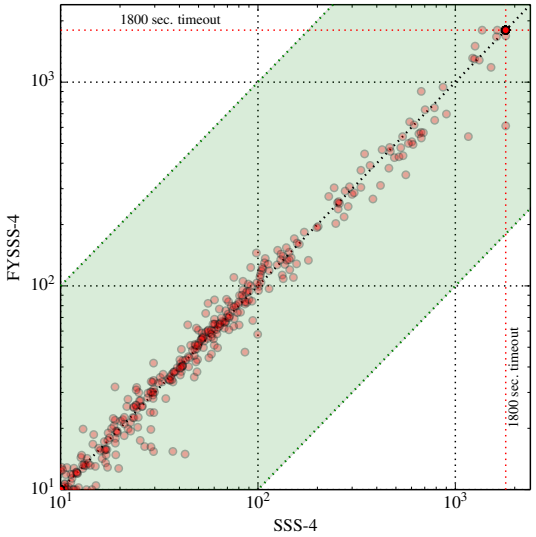

Figure 5.8: Comparison between run times of the basic (SSS-4) and fully incremental (FYSSS-4) versions of the Search Space Splitting with 4 processes, in the same machine, for industrial instances.

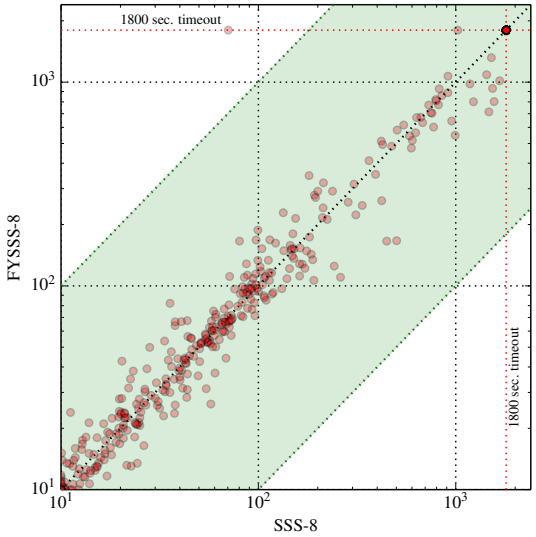

Figure 5.9: Comparison between run times of the basic (SSS-8) and fully incremental (FYSSS-8) versions of the Search Space Splitting with 8 processes, in the same machine, for industrial instances.

5.2 Guiding Paths with Lookahead Algorithm

Figures 5.10 and 5.11 show the run times of the Guiding Paths algorithm (GP-2:4) with 8 processes, 2 per machine, compared with OpenWBO-LS, for crafted and industrial instances respectively. We can clearly see that GP-2:4 is much more effective in the crafted category than in the industrial category. As stated in section 4.2, lookahead solvers for SAT are known to be not very effective in industrial instances. The guiding path generation generator (algorithm 7) works in the same way as a lookahead solver, the difference being that the generator partitions the search tree instead of looking for a solution. Therefore, it was expected that the Guiding Paths algorithm would be much more effective in crafted instances than in industrial instances.

GP-2:4 outperforms OpenWBO-LS in most crafted instances. In fact, we can see in figure 5.10 various instances that GP-2:4 managed to solve around 10 times faster than OpenWBO-LS. However, there are also a few instances on which GP-2:4 was around 10 times slower than OpenWBO-LS. This can happen if the guiding paths generated by algorithm 7 force the SAT solver to traverse branches of the search tree that are hard to solve and that would not be traversed if the search tree had not been partitioned. Therefore, a lot can be gained in performance by using a Guiding Paths with Lookahead algorithm, but it depends on which guiding paths are being generated. It would be interesting to determine if there exists heuristics for guiding path generation that would result in an improvement of the performance in the industrial category at the cost of the performance in the crafted category.

Figures 5.12 and 5.13 show the run times of GP-2:4 compared with the run times of SSS-2:4. The results are very similar to what was observed in figures 5.10 and 5.11. GP-2:4 is clearly outperformed by SSS-2:4 in the industrial category. However, GP-2:4 is slightly more effective in the crafted category than SSS-2:4, solving 1 more instance.

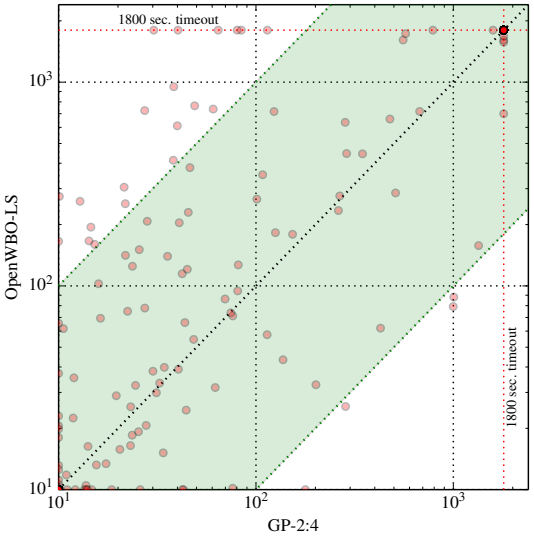

Figure 5.10: Comparison between run times of the Guiding Paths algorithm (GP-2:4) with 8 processes, in 4 machines, and OpenWBO's Linear Search (OpenWBO-LS) for crafted instances.

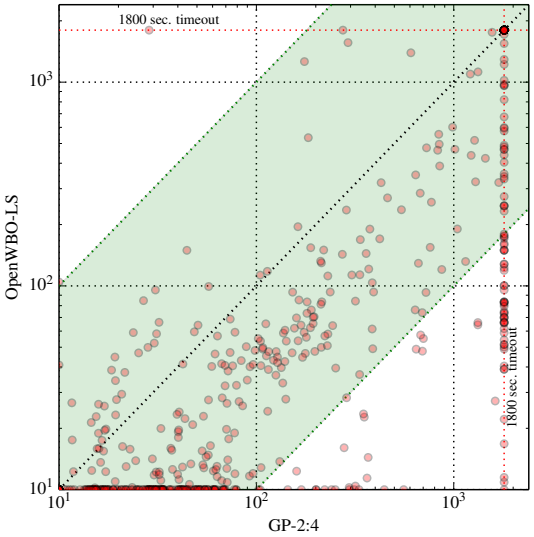

Figure 5.11: Comparison between run times of the Guiding Paths algorithm (GP-2:4) with 8 processes, in 4 machines, and OpenWBO's Linear Search (OpenWBO-LS) for industrial instances.

We can easily conclude that the Search Space Splitting algorithm, as described in section 4.1, is more effective in the industrial category, and that the Guiding Paths with Lookahead algorithm, as described in section 4.2, is more effective in the crafted instances.

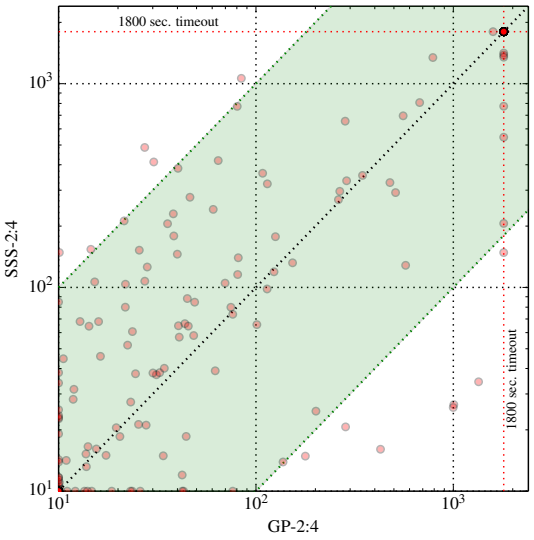

Figure 5.12: Comparison between run times of the Guiding Paths algorithm (GP-2:4) and Search Space Splitting with 8 processes, in 4 machines, for crafted instances (SSS-2:4).

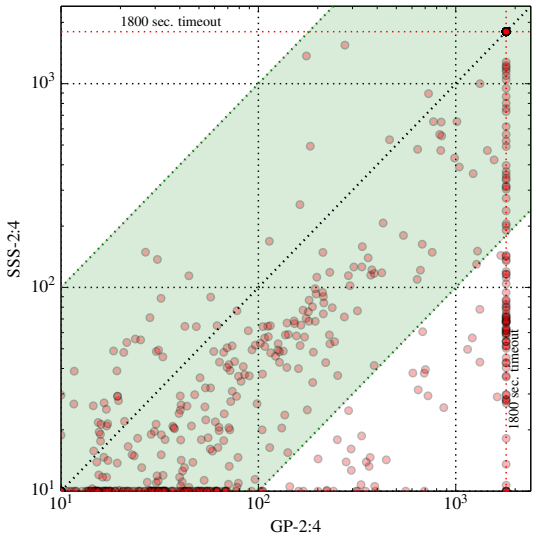

Figure 5.13: Comparison between run times of the Guiding Paths algorithm (GP-2:4) and Search Space Splitting with 8 processes, in 4 machines, for industrial instances (SSS-2:4).

Chapter 6

Conclusions

MaxSAT is a NP-hard problem with many applications in the real world. Several sequential algorithms have been proposed and improved in the recent years, making it possible to solve many instances of interest quite fast. Still, there exist instances that are very hard to solve in reasonable time, and so much work has been done on MaxSAT so far that it becomes harder to further improve sequential MaxSAT solving. For this reason, some parallel algorithms have been proposed and studied, but this kind of algorithms has very limited scalability due to BUS contention on access to primary memory. Also, a machine becomes much more costly as the number of processor cores increases.

Distributed algorithms are more likely to scale than parallel algorithms, as long as the amount of communication required between threads is not heavy, since threads are spread among computation nodes, each with its own independent primary memory. For this reason, we implemented two different distributed approaches for solving MaxSAT and studied how these approaches fare in comparison with one of the most effective sequential MaxSAT solvers that exist nowadays. We also studied how both distributed algorithms fare in comparison with each other.

6.1 Achievements

This dissertation introduced the first two distributed algorithms for MaxSAT. The first algorithm is based on splitting among processes the interval of possible objective values of the optimum solution and shrinking this interval until an optimum solution is found (section 4.1). One process searches on a lower bound, another on an upper bound and the others on local upper bounds. Two versions of this algorithm were implemented and tested. This approach was shown to be effective in practice at speeding up the search process, both in the crafted and industrial instances. Therefore, distributing algorithms can improve the performance of MaxSAT solvers.

The second algorithm partitions the search tree and assigns different sub-trees to distinct processes (section 4.2). This algorithm is based on an approach proposed for SAT solving that was shown to improve the performance of sequential, parallel and distributed SAT solvers. This algorithm was shown to be very effective in the crafted instances, but was not shown to be competitive in the industrial instances.

6.2 Future Work

In the future, both algorithms should be further tested in order to better evaluate their scalability. It would be ideal to obtain results with more processes spread across more machines in order to determine if these distributed approaches are scalable or not.

Both algorithms were designed with modularity in mind. One can choose among multiple SAT solvers to be used by the distributed algorithm, as long as they implement the corresponding interface. However, this feature has the drawback of forcing processes to wait for each SAT invocation to terminate before checking for new messages. Changing the SAT solver and having it checking for messages during its search process could further improve the distributed algorithms by terminating a SAT invocation when it is no longer needed, after notification from the mediator.

Implementing clause sharing would also imply changing the SAT solver. It would be interesting to see how clause sharing would influence the performance of both distributed algorithms.

There is also much investigation that could still be done on the Guiding Paths algorithm of section 4.2. More thorough testing should be carried with different heuristics and with different values for the various configurable parameters of the algorithm. Different values may improve the performance of MaxSAT in comparison to SAT, and different heuristics may be more suitable for solving industrial instances than the ones currently tested.

Finally, one should improve the runsolver application, used to run the implemented algorithms in a controlled environment, in order for it to function properly when dealing with distributed solvers.

Bibliography

T. Alsinet, F. Manyà, and J. 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.

T. Alsinet, F. Manyà, and J. Planes. A Max-SAT Solver with Lazy Data Structures. In *Proceedings of the 9th Ibero-American Conference on Artificial Intelligence*, pages 334–342. Springer, 2004.

T. Alsinet, F. Manyà, and J. Planes. Improved Exact Solvers for Weighted Max-SAT. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, pages 371–377. Springer, 2005.

T. Alsinet, F. Manyà, and J. Planes. An efficient solver for weighted Max-SAT. *Journal of Global Optimization*, 41:61–73, 2008.

X. An, M. Koshimura, H. Fujita, and R. Hasegawa. QMaxSAT version 0.3 & 0.4. In *Workshop on First-Order Theorem Proving*, pages 7–15, 2011.

X. An, T. Zhang, H. Fujita, and R. Hasegawa. QMaxSAT: A Partial Max-SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:95–100, 2012.

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

C. Ansótegui, M. L. Bonet, and J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 427–440, 2009a.

C. Ansótegui, M. L. Bonet, and J. Levy. On Solving MaxSAT Through SAT. In *International Conference of the Catalan Association for Artificial Intelligence*, pages 284–292, 2009b.

C. Ansótegui, M. L. Bonet, and J. Levy. A New Algorithm for Weighted Partial MaxSAT. In *AAAI Conference on Artificial Intelligence*, pages 3–8, 2010.

R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality Networks: a theoretical and empirical study. *Constraints*, 16:195–221, 2011.

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

G. Audemard, J.-M. Lagniez, and L. Simon. Improving glucose for incremental sat solving with assumptions: application to mus extraction. In *Theory and Applications of Satisfiability Testing–SAT 2013*, pages 309–317. Springer, 2013.

O. Baileux and Y. Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In *Proc. International Conference on Principles and Practice of Constraint Programming*, pages 108–122. Springer, 2003.

O. Bailleux, Y. Boufkhad, and O. Roussel. A Translation of Pseudo-Boolean Constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:191–200, 2006.

N. Bansal and V. Raman. Upper Bounds for MaxSat: Further Improved. In *Proc 10th International Symposium on Algorithms and Computation*, pages 247–260. Springer, 1999.

K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314. ACM, 1968.

M. Böhm and E. Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.

B. Borchers and J. Furman. A Two-Phase Exact Algorithm for MAX-SAT and Weighted MAX-SAT Problems. *Journal of Combinatorial Optimization*, 2:299–306, 1999.

M. Büttner and J. Rintanen. Satisfiability Planning with Constraints on the Number of Actions. In *International Conference on Automated Planning and Scheduling*, pages 292–299. AAAI Press, 2005.

B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial MaxSAT. In *AAAI Conference on Artificial Intelligence / IAAI Innovative Applications of Artificial Intelligence Conference*, pages 263–268, 1997.

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

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

M. Codish, V. Lagoon, and P. J. Stuckey. Logic Programming with Satisfiability. *Journal of Theory and Practice of Logic Programming*, 8:121–128, 2008.

S. Darras, G. Dequen, L. Devendeville, and C.-M. Li. On Inconsistent Clause-Subsets for Max-SAT Solving. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 225–240. Springer, 2007.

M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5:394–397, 1962.

N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.

A. Frisch, T. Peugniez, A. Doggett, and P. Nightingale. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings. *Journal of Automated Reasoning*, 35:143–179, 2005.

Z. Fu and S. Malik. On Solving the Partial MAX-SAT Problem. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 252–265, 2006.

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

E. Giunchiglia and M. Maratea. Planning as Satisfiability with Preferences. In *AAAI Conference on Artificial Intelligence*, pages 987–992, 2007.

Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009a.

Y. Hamadi, S. Jabbour, and L. Sais. Control-Based Clause Sharing in Parallel SAT Solving. In *Proc. International Joint Conferences on Artificial Intelligence*, pages 499–504. IJCAI/AAAI Press, 2009b.

F. Heras and J. Larrosa. New Inference Rules for Efficient Max-SAT Solving. In *Proceedings of the National Conference on Artificial Intelligence*, pages 68–73, 2006.

F. Heras, J. Larrosa, and A. Oliveras. MiniMaxSAT: An Efficient Weighted Max-SAT Solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.

F. Heras, A. Morgado, and J. Marques-Silva. Core-Guided Binary Search Algorithms for Maximum Satisfiability. In *AAAI Conference on Artificial Intelligence*, pages 36–41, 2011.

M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In *Hardware and Software: Verification and Testing*, pages 50–65. Springer, 2012.

S. Hülldoble and V. H. Nguyen. An Efficient Encoding of the at-most-one Constraint. Technical Report KRR 13-04, Technische Universität Dresden, Faculty of Computer Science, Institute of Artficial Intelligence, Knowledge Representation and Reasoning, 2013.

R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *ACM SIGPLAN Notices*, volume 46, pages 437–446. ACM, 2011.

W. Klieber and G. Kwon. Efficient CNF Encoding for Selecting 1 from N Objects. In *International Workshop on Constraints in Formal Verification*, 2007.

J. Larrosa and F. Heras. Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 193–198, 2005.

J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient Max-SAT solving. *Artificial Intelligence*, 172:204–233, 2008.

D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

C. M. Li, F. Manyà, and J. 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*, pages 403–414. Springer, 2005.

C. M. Li, F. Manyà, and J. Planes. Detecting Disjoint Inconsistent Subformulas for Computing Lower Bounds for Max-SAT. In *Proceedings of the 21st National Conference on Artificial Intelligence*, pages 86–91, 2006.

C. M. Li, F. Manyà, and J. Planes. New Inference Rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

H. Lin and K. Su. Exploiting Inference Rules to Compute Lower Bounds for MAX-SAT Solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2334–2339, 2007.

H. Lin, K. Su, and C. M. Li. Within-problem Learning for Efficient Lower Bound Computation in Max-SAT Solving. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, volume 8, pages 351–356, 2008.

H. Mangassarian, A. Veneris, S. Safarpour, F. N. Najm, and M. S. Abadir. Maximum circuit activity estimation using pseudo-boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1538–1543. EDA Consortium, 2007.

V. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for Weighted Boolean Optimization. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 495–508, 2009.

J. Marques-Silva and V. Manquinho. Towards More Effective Unsatisfiability-Based Maximum Satisfiability Algorithms. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 225–230, 2008.

J. Marques-Silva and J. Planes. On Using Unsatisfiability for Solving Maximum Satisfiability. *CoRR*, abs/0712.1097, 2007.

J. Marques-Silva and J. Planes. Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In *Conference on Design, Automation and Testing in Europe*, pages 408–413, 2008.

J. Marques-Silva and K. Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, 1996.

J. Marques-Silva, I. Lynce, and S. Malik. Conflict-Driven Clause Learning SAT Solvers. *SAT Handbook*, pages 131–154, 2009.

R. Martins. *Parallel Search for Maximum Satisfiability*. PhD thesis, Instituto Superior Técnico, 2013.

R. Martins, V. Manquinho, and I. Lynce. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *Proc. International Conference on Tools with Artificial Intelligence*, pages 313–320. IEEE Computer Society Press, 2011a.

R. Martins, V. Manquinho, and I. Lynce. Parallel Search for Boolean Optimization. In *RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion*, 2011b.

R. Martins, V. Manquinho, and I. Lynce. Clause Sharing in Parallel MaxSAT. In *Proc. Learning and Intelligent Optimization Conference*, pages 455–460. Springer, 2012a.

R. Martins, V. Manquinho, and I. Lynce. Parallel Search for Maximum Satisfiability. *AI Communications*, 25:75–95, 2012b.

R. Martins, S. Joshi, V. Manquinho, and I. Lynce. Incremental cardinality constraints for maxsat. In *Principles and Practice of Constraint Programming*, pages 531–548. Springer, 2014a.

R. Martins, V. Manquinho, and I. Lynce. Open-wbo: a modular maxsat solver. In *Theory and Applications of Satisfiability Testing–SAT 2014*, pages 438–445. Springer, 2014b.

A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva. Iterative and Core-Guided MaxSAT Solving: A Survey and Assessment. *Constraints*, 18:478–534, 2013.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

R. Niedermeier and P. Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36:63–88, 2000.

K. Pipatsrisawat, A. Palyan, M. Chavira, A. Choi, and A. Darwiche. Solving Weighted Max-SAT Problems in a Reduced Search Space: A Performance Analysis. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:191–217, 2008.

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

M. Ramírez and H. Geffner. Structural Relaxations by Variable Renaming and Their Compilation for Solving MinCostSAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 605–619. Springer, 2007.

O. Roussel. Controlling a solver execution with the runsolver tool system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.

T. Schubert, M. Lewis, and B. Becker. PaMiraXT: Parallel SAT Solving with Threads and Message Passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.

H. Shen and H. Zhang. Study of Lower Bound Functions for MAX-2-SAT. In *Proceedings of the 19th National Conference on Artificial Intelligence*, pages 185–190, 2004.

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

D. M. Strickland, E. Barnes, and J. S. Sokol. Optimal Protein Structure Alignment Using Maximum Cliques. *Operations Research*, 53:389–402, 2005.

R. Wallace and E. Freuder. Comparative studies of constraint satisfaction and davis-putnam algorithms for maximum satisfiability problems. *Cliques, Coloring and Satisfiability*, 26:587–615, 1996.

J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68:63–69, 1998.

Z. Xing and W. Zhang. Efficient Strategies for (Weighted) Maximum Satisfiability. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 690–705. Springer, 2004.

Z. Xing and W. Zhang. MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence*, 164:47–80, 2005.

H. Xu, R. A. Rutenbar, and K. Sakallah. sub-SAT: a formulation for relaxed Boolean satisfiability with applications in routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22:814–820, 2003.

H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *International Conference on Computer-Aided Design*, pages 279–285, 2001.

Appendix A

Cardinality Constraint Encodings

It is relevant to consider the special case of at-most-k constraints where $k = 1$. These are called *at-most-1 constraints* and some of the proposed encodings are designed and optimized just for that kind of constraints. A brief description of encodings is given now for at-most-1 constraints. The literature for those encodings is also provided in case the reader is interested in looking for further details. Throughout the following descriptions we are always considering an arbitrary set of Boolean variables $X = \{x_1, ..., x_n\}$ and an at-most-1 constraint given by $\sum_{i=1}^{n} x_i \leq 1$.

- *Pairwise*: let ϕ be a new empty set of clauses. For each pair of variables (x_i, x_j) in the at-most-1 constraint, where $i \neq j$, the clause $(\neg x_i \vee \neg x_j)$ is added to ϕ. This new clause will be unsatisfied if both variables are assigned the truth value 1, thus forcing one of the variables to be 0. The resulting ϕ is the *pairwise encoding* for the constraint.

- *Ladder* (Gent and Nightingale [2004], Ansótegui and Manyà [2004]): the *ladder encoding* uses a sequence of n auxiliary variables $y_1, ..., y_n$, ordered by index, referred to as the *ladder variables*. A variable y_i can only be assigned the truth value 1 if all other variables y_j such that $j < i$ are also assigned 1. Consequently, a variable y_i can only be assigned the truth value 0 if all other variables y_j such that $j > i$ are assigned the truth value 0. The ladder variables coupled with the aforementioned constraints, referred to as *ladder validity constraints*, define a structure called the *ladder*. The ladder has $n + 1$ valid states[1]. The first n states are associated with the variables in X in an one-to-one relationship as follows: if x_i is assigned truth value 1, then y_i must be the first variable assigned 0 in the sequence of ladder variables, and vice versa. The remaining state (all ladder variables assigned to 1) is used to encode the possibility of all variables in X being assigned the truth value 0.

- *Bitwise* (Frisch et al. [2005], Prestwich [2007]): the *bitwise encoding* uses a sequence of $\lceil \log_2 n \rceil$ auxiliary variables $y_1, ..., y_{\lceil \log_2 n \rceil}$. This sequence forms a bit string with $\lceil \log_2 n \rceil$ bits, and this string is used to encode the indexes of the variables in X. A new clause $(\neg x_i \vee y_j)$ is added, to the new CNF formula for the at-most-1 constraint, for each pair (x_i, y_j) such that bit j of the binary

[1] In other words, there exist $n + 1$ assignments to the ladder variables that satisfy the ladder validity constraints.

representation of $i-1$ is 1. Otherwise, if bit j of the binary representation of $i-1$ is 0, then the clause $(\neg x_i \vee \neg y_j)$ is added instead. Note that if n is not a power of 2 then there will be more bit strings than the variables in X. In this case, either the number of possible different bit strings must be reduced to n, or the extra bit strings must be associated with some of the variables in X. A solution was proposed by Frisch et al. [2005] that does not require additional clauses.

- *Commander* (Klieber and Kwon [2007]): the set X is divided into disjoint subsets $G_1, ..., G_m$, usually of size 3, and, for every subset G_k, an auxiliary *commander variable* c_k is associated with G_k. To build the *commander encoding* for the at-most-1 constraint, we start by adding clauses to the new CNF formula that encode the fact that only one variable in each subset can be assigned truth value 1 (the pairwise encoding can be used for each subset). Then, for each subset G_k, the clause $(\neg c_k \vee \bigvee_{x_i \in G_k} x_i)$ is added, forcing the variables in G_k to be assigned the truth value 0 if variable c_k is assigned 0. Lastly, it must be encoded that no more than one of the commander variables can be assigned the truth value 1 in the same assignment, this can be done either by using one of the other encodings for at-most-1 constraints or by applying the commander encoding recursively on the commander variables. A mixture of this encoding with the aforementioned bitwise encoding that uses bit strings to represent the *commander variables* was proposed by Hülldoble and Nguyen [2013].

- *Product* (Chen [2010]): the *product encoding* uses two disjoint sets of auxiliary variables $U = \{u_1, ..., u_p\}$ and $V = v_1, ..., v_q$, where $p = \lceil \sqrt{n} \rceil$ and $q = \lceil \frac{n}{p} \rceil$. A pair (u_i, v_j) ($u_i \in U$ and $v_j \in V$) is associated with the coordinates (i, j) of a $p \times q$ grid. Each variable $x_k \in X$ is assigned to one position in that grid, in a one-to-one relationship. The new CNF formula for the at-most-1 constraint on the variables in X is constructed by adding one at-most-1 constraint on the variables in U[2], one at-most-1 constraint on the variables in V and clauses that associate each variable $x_k \in X$ with a pair (u_i, v_j) such that $u_i \in U$ and $v_j \in V$, forcing u_i and v_j to be assigned the truth value 1 if x_k is assigned 1. In the case where there are more grid positions than variables in X ($n < p \times q$) the extra grid positions are simply ignored.

Now we move on to encodings for the general at-most-k constraints. The same set of variables X from before is considered throughout the following descriptions, but this time the constraint to be encoded has the form $\sum_{i=1}^{n} x_i \leq k$.

- *Sequential Counter* (Sinz [2005]): the constraint is encoded as a Boolean circuit that takes the variables in X as input and counts in unary how many of those variables are assigned truth value 1. This circuit is composed by a sequence of n smaller circuits. The i-th circuit in this sequence computes the partial sum $\sum_{j=1}^{i} x_j$, given x_i and a sequence of variables $s_{i-1,1}, ..., s_{i-1,k}$, that encode the partial sum computed by the $(i-1)$-th circuit[3]. The CNF formula that represents the *sequential encoding* for the constraint is composed by clauses that encode the aforementioned

[2]Any of the encodings described can be used for this effect.

[3]Note that the first circuit in the sequence will only receive x_1 as input since it is the first variable to be considered in the counting process.

Boolean circuit and clauses that force each variable $x_i \in X$ to be assigned 0 if the sequence of variables $s_{i-1,1}, ..., s_{i-1,k}$ is the unary representation of k. A variant of this encoding is presented in the literature (Sinz [2005]) that uses a *parallel counter* instead. This variant recursively divides the set X into two halves and computes the partial sum of those subsets in binary.

- *Totalizer* (Baileux and Boufkhad [2003]): this encoding is based on the sum in unary. This encoding's structure can be seen as a binary tree, where the leaves are the variables in the set X in a one-to-one relationship. The intermediate nodes (including the root) represent the sum in unary of its leaves. So, given a sub-tree such that its leaves are the variables $x_i, ..., x_j$, its root is represented by a sequence of i Boolean auxiliary variables that encode the sum $\sum_{k=i}^{j} x_k$ in unary. Therefore, the root of the main binary tree is represented by n auxiliary variables that encode the sum $\sum_{i=1}^{n} x_i$ in unary. The partial sum at each intermediate node is computed from the partial sums of its children nodes. The CNF formula for the *totalizer encoding* is composed by clauses that encode the aforementioned binary tree and clauses that force the unary sum at the root to be less than or equal to k. For example, given a sequence of auxiliary variables $s_1, ..., s_n$, the last condition can be achieved with the set of clauses $\{(\neg s_{k+1}), ..., (\neg s_n)\}$. This version of the totalizer encoding counts up to n, but Büttner and Rintanen [2005] proposed a variant that only counts up to $k + 1$, reducing the number of clauses.

- *Sorters* (Eén and Sörensson [2006], Batcher [1968]): this encoding uses a Boolean circuit that sorts the inputs given by the variables $x_1, ..., x_n$. The circuit's output is the sequence of auxiliary Boolean variables $y_1, ..., y_n$. Because $y = y_1...y_n$ is the result of sorting the set $X = \{x_1, ..., x_n\}$, y is the unary representation of the number of variables in X that are assigned the truth value 1. Therefore, we can force that number to be less than or equal to k in the same way as in the totalizer encoding. The initial sorting of two variables x_i and x_j is achieved by computing the minimum and the maximum of the pair $\{x_i, x_j\}$. The minimum is equivalent to computing $(x_i \wedge x_j)$ and the maximum is equivalent to $(x_i \vee x_j)$. This results in $\frac{n}{2}$ ordered sets of Boolean values. At each step, the circuit merges two ordered subsets into a single set that is also ordered. This process is repeated until only one ordered set remains, which corresponds to the variables $y_1, ..., y_n$. Optimizations proposed for the sorters encoding can be found in the literature (Asín et al. [2011], Codish and Zazon-Ivry [2010]).

All the encodings for at-most-1 constraints require $O(n)$ clauses and $O(n)$ auxiliary variables, except for the bitwise encoding, which requires $O(n\ log_2 n)$ clauses and $O(log_2 n)$ auxiliary variables, and the pairwise encoding, which requires $O(n^2)$ clauses and no auxiliary variables. In the case of at-most-k constraints, all of them require $O(nk)$ clauses, except for the sorters encoding that requires $O(n\ log_2^2 n)$. The sequential encoding requires $O(nk)$ auxiliary variables, totalizer requires $O(n\ log_2 n)$ and sorters requires $O(n\ log_2^2 n)$. An experimental evaluation of all these encodings was done by Martins et al. [2011a, 2012b].