Master's thesis

Master's Programme in Computer Science

Refined Core Relaxations for Core-Guided Maximum Satisfiability Algorithms

Hannes Ihalainen

November 21, 2022

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Master's Programme in Computer Science

Tekijä — Författare — Author

Hannes Ihalainen

Työn nimi — Arbetets titel — Title

Refined Core Relaxations for Core-Guided Maximum Satisfiability Algorithms

Ohjaajat — Handledare — Supervisors

Professor Matti Järvisalo, Docent Jeremias Berg

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year November 21, 2022	Sivumäärä — Sidoantal — Number of pages 65 pages

Tiivistelmä — Referat — Abstract

The so-called declarative approach has proven to be a viable paradigm for solving various real-world NP-hard optimization problems in practice. In the declarative approach, the problem at hand is encoded using a mathematical constraint language, and an algorithm for the specific language is employed to obtain optimal solutions to an instance of the problem. One of the most viable declarative optimization paradigms of the last years is maximum satisfiability (MaxSAT) with propositional logic as the constraint language.

So-called core-guided MaxSAT algorithms are arguably one of the most effective MaxSAT-solving paradigms in practice today. Core-guided algorithms iteratively detect and rule out (relax) sources of inconsistencies (so-called unsatisfiable cores) in the instance being solved. Especially effective are recent algorithmic variants of the core-guided approach which employ so-called soft cardinality constraints for ruling out inconsistencies.

In this thesis, we present a structure-sharing technique for the cardinality-based core relaxation steps performed by core-guided MaxSAT solvers. The technique aims at reducing the inherent growth in the size of the propositional formula resulting from the core relaxation steps. Additionally, it enables more efficient reasoning over the relationships between different cores.

We empirically evaluate the proposed technique on two different core-guided algorithms and provide open-source implementations of our solvers employing the technique. Our results show that the proposed structure-sharing can improve the performance of the algorithms both in theory and in practice.

ACM Computing Classification System (CCS)
Theory of computation → Logic → Constraint and logic programming
Mathematics of computing → Discrete mathematics → Combinatorics → Combinatorial optimization

Avainsanat — Nyckelord — Keywords

declarative optimization, maximum satisfiability, core-guided search, cardinality constraints

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms study track

Contents

1 Introduction

NP-hard optimization problems arise in many real-world applications [16, 66, 81]. Hence there is a need for developing practically efficient generic algorithmic approaches that can be employed for finding optimal solutions to such problems.

One successful approach for solving computationally hard real-world optimization problems is the so-called declarative approach. In the declarative approach, the problem instance at hand is encoded in a mathematical constraint language. An encoding allows for expressing any instance of the original problem in the mathematical constraint language in a way that the optimal solutions to the original problem instance correspond to the optimal solutions to the encoded instance. An algorithm for the constraint language is employed to obtain an optimal solution for the encoded instance. An optimal solution to the original problem instance can then be reconstructed from the optimal solution to the encoded instance. An implementation of an algorithm developed for a specific declarative language is called a *solver*. Given an instance expressed in the language a solver supports and enough resources, the solver is guaranteed to obtain an optimal solution to the instance.

An advantage of the declarative approach is that, when faced with a new problem, there is no need to develop a new practically efficient solver. Instead, it is sufficient to develop an encoding for the problem instances in a constraint language for which practically efficient solvers are already available. Another advantage is that when a same constraint language is used for encoding various real-world domains, improvements in solvers for the language have an immediate impact on the efficiency of solving instances from the various problem domains.

Arguably the most classical declarative solving paradigm for optimization problems is integer programming (IP) [31, 81]. Other declarative paradigms include constraint optimization problem (COP) [74], pseudo-Boolean optimization (PBO) [29], maximum satisfiability (MaxSAT) [16, 51], optimization on SAT modulo theories (MaxSMT) [70] and answer-set programming (ASP) [41]. This thesis focuses in particular on MaxSAT as one of the most viable paradigms of recent years.

The constraint language of MaxSAT is that of propositional logic [16, 51]. A MaxSAT instance consists of a CNF formula (a propositional formula in conjunctive normal form) and a linear objective function over Boolean variables to be minimized [16, 51].[1] MaxSAT is an optimization variant of the classical Boolean satisfiability problem (SAT) [32, 50], where the task is to decide if there exists an assignment that satisfies a given propositional formula. Over the last two decades, MaxSAT has been successfully applied in many real-world applications, including planning, scheduling, configuration, artificial intelligence, data analysis, combinatorial problems, verification and security, software analysis and bioinformatics among others. A recent survey of applications and relevant references can be found in [16].

Many different practical algorithms have been proposed for MaxSAT [16]. Empirical evaluations have shown that the empirically most efficient approach depends on the instance at hand [17]. Practical algorithms for MaxSAT can be divided to complete and incomplete algorithms. Incomplete (or "anytime") algorithms seek to find a good (not necessarily an optimal) solution in a limited time whereas complete (or "exact") algorithms are guaranteed to find an optimal solution when given enough resources. Most of the modern complete algorithms for MaxSAT are so-called Boolean satisfiability (SAT) based approaches where the optimization problem is solved by reducing it to a sequence of decision problems [16]. The success of this approach builds heavily on the success of the development of practically efficient SAT solvers [16]. The most central SAT-based approaches for MaxSAT are the so-called model-improving [16, 1], core-guided [16, 39] and implicit hitting set (IHS) approaches [16, 34].

This work focuses on the core-guided approach. The core-guided approach does a lower-bounding search for obtaining an optimal solution. It uses a SAT solver iteratively to find inconsistencies (so-called *cores*) in the CNF formula that incur cost. The cores are then relaxed by doing transformations on the formula. One drawback in the efficiency of the core-guided approach is that the core transformations increase the size of the CNF formula and the blow-up in the size can make the formula increasingly hard for a SAT solver to solve [34, 16, 19].

The main contribution of this work is a novel structure-sharing technique that reduces the inherent blow-up in the size of the formula on weighted MaxSAT instances. Structure-sharing employs the fact that the transformations of modern core-guided algorithms add

[1]Even though the name MaxSAT refers to a maximization problem (maximize the number of satisfied clauses/the sum of their weights), it has developed standard to see MaxSAT as the equivalent minimization problem (minimize the number of unsatisfied clauses/the sum of their weights) [51, 16]. Without loss of generality, we view the optimization of MaxSAT as minimizing a linear function over the variables of the instance as will be explained in Chapter 2.

to the formula so-called soft cardinality constraints over the cores [65, 69, 16]. When the cores overlap, cardinality constraint encodings introduce redundant clauses and variables to the constraints. In such a case, it is possible to reduce the size of the encoding by sharing some parts between different cardinality constraint encodings rather than encoding every cardinality constraint separately from scratch.

In this thesis, we describe how to incorporate structure-sharing into two modern core-guided algorithms, OLL [65, 3] and PMRES [69], to reduce the size of the cardinality constraint encodings they use. Our approach relies heavily on the recently-proposed weight-aware core extraction technique (WCE) [23] that enables relaxing a number of cores on each iteration. In particular, WCE allows making informed decisions about which substructures of cardinality constraints to share. Interestingly, our proposed technique not only reduces the size of the formula but also improves the ability of a SAT solver to do reasoning about the relationships between cores. We provide open-source implementations of solvers employing the structure-sharing technique and show by empirical evaluation that our implementations outperform a state-of-the-art MaxSAT solver. Even though this thesis concentrates on the two mentioned core-guided algorithms, the idea of structure-sharing could also be incorporated into other core-guided algorithms and even into other approaches (e.g., into implicit hitting set approach [16, 34] with so-called abstract cores [21]) or into incomplete solving [22].

The main contribution of this thesis—the structure-sharing technique—has been published at the *27th International Conference on Principles and Practice of Constraint Programming* (CP 2021) [46]. In this thesis, we extend the results of [46] by providing a more thorough discussion of the new techniques and by providing a novel C++ implementation of our algorithm which we show to be more efficient in practice than the implementation empirically evaluated in [46].

The structure of the thesis is as follows. In Chapter 2 we introduce definitions of the important concepts necessary to understand the main contribution of this thesis. The two core-guided algorithms into which we incorporate the structure-sharing are detailed in Chapter 3. In Chapter 4 we introduce our main contribution, the structure-sharing technique. Finally, we discuss our implementation and empirical results in Chapter 5.

2 Preliminaries

In this chapter, we overview the Boolean satisfiability problem (SAT) and maximum satisfiability (MaxSAT). We provide definitions for the central concepts necessary for understanding our contributions and introduce the notation we use throughout the thesis. We also put our work in a larger context by providing an overview of different modern algorithms for MaxSAT.

2.1 Satisfiability

A *literal* l is either a Boolean variable x or its negation $\neg x$. Logical negation extends naturally to literals: $\neg l = \neg x$ if $l = x$ and $\neg l = x$ if $l = \neg x$. A *clause* is a disjunction $C = l_1 \vee \cdots \vee l_n)$ of literals l_i. A CNF *formula* (a propositional formula in conjunctive normal form) is a conjunction $C_1 \wedge \cdots \wedge C_m$ of clauses.

A *truth assignment* τ assigns Boolean values—either 1 (true) or 0 (false)—to a subset of the variables of a CNF formula $\mathcal{F}$. A truth assignment extends naturally to literals by $\tau(\neg x) = 1 - \tau(x)$. When convenient, we treat τ as the set of literals it assigns to 1. A truth assignment τ is *complete* with respect to a CNF formula $\mathcal{F}$ if τ assigns all variables that occur in $\mathcal{F}$; otherwise, τ is *partial*. A truth assignment τ falsifies a clause C if τ assigns all literals in C to 0, and satisfies C if τ assigns at least one literal in C to 1. A truth assignment τ falsifies $\mathcal{F}$ if τ falsifies one clause in $\mathcal{F}$, and satisfies $\mathcal{F}$ if τ satisfies all clauses in $\mathcal{F}$. A satisfying assignment of $\mathcal{F}$ is a *solution* to $\mathcal{F}$. If there is a satisfying assignment for $\mathcal{F}$ then $\mathcal{F}$ is *satisfiable*, otherwise $\mathcal{F}$ is *unsatisfiable*.

Example 2.1

Consider the CNF formula $\mathcal{F} = (x \vee \neg y) \wedge (\neg x \vee \neg y) \wedge (y \vee \neg z)$. The assignment $\tau_1 = \{\neg x, y, \neg z\}$ is a falsifying complete assignment of $\mathcal{F}$. It falsifies the first clause $(x \vee \neg y)$, thus falsifying $\mathcal{F}$. The assignment $\tau_2 = \{\neg y, \neg z\}$ is a satisfying partial assignment. The partial assignment $\tau_3 = \{\neg y\}$ is neither a satisfying nor falsifying assignment. While it satisfies the first two clauses, the clause $(y \vee \neg z)$ is neither satisfied nor falsified.

Example 2.2

The CNF formula $\mathcal{F} = (x \vee y) \wedge (\neg x \vee y) \wedge (\neg y)$ is unsatisfiable. To see this, note that any satisfying assignment should include $\neg y$ in order to satisfy $(\neg y)$. However, the assignment $\tau_1 = \{x, \neg y\}$ falsifies the clause $(\neg x \vee y)$ and the assignment $\tau_2 = \{\neg x, \neg y\}$ falsifies the clause $(x \vee y)$. Therefore there is no assignment that would satisfy all three clauses in $\mathcal{F}$.

The CNF formula $\mathcal{F} = (x \vee y) \wedge (\neg x \vee \neg y) \wedge (x)$ is satisfiable: the assignment $\tau = \{x, \neg y\}$ satisfies $\mathcal{F}$.

The classical NP-complete Boolean satisfiability problem (SAT) [32, 50, 40, 11] asks to decide if a given propositional formula is satisfiable or unsatisfiable. As typical, we assume that propositional formulas are in CNF. It is well-known that any propositional formula can be transformed into a CNF formula of linear size with respect to the original formula with the standard Tseitin encoding [79, 73].

A *SAT solver* is an implementation of a decision procedure for SAT, i.e., an algorithm that, given a CNF formula $\mathcal{F}$ (and enough resources), decides if $\mathcal{F}$ is satisfiable or unsatisfiable. Modern solvers also provide a satisfying assignment if $\mathcal{F}$ is satisfiable. The most important algorithmic approach employed in SAT solvers today is the so-called conflict-driven clause learning (CDCL) SAT solving paradigm [57]. CDCL SAT solvers maintain a partial assignment over the variables of $\mathcal{F}$ that is extended towards a complete assignment by iteratively assigning values to the variables in a backtracking style search. The aim is to form a satisfying complete assignment to $\mathcal{F}$ (proving $\mathcal{F}$ is satisfiable) or to determine that $\mathcal{F}$ is unsatisfiable. More concretely, the following steps are performed iteratively.

- Make a *decision*. Assign some unassigned variable (selected by heuristics) to either 1 or 0.

- Perform *unit propagation* until fixpoint. Unit propagation finds clauses in which all literals except one have been assigned to 0 by the current partial assignment, and the one remaining literal is unassigned. In order to satisfy the clause, the currently unassigned literal is deterministically assigned to 1.

- If a *conflict* is detected (some clause is falsified by the current partial assignment) the algorithm learns a so-called conflict clause over a subset of the currently assigned variables, adds the clause to $\mathcal{F}$ and backtracks non-chronologically [57, 61] (i.e., unassigns variables until one literal in the conflict clause is unassigned). A conflict clause is logically entailed by $\mathcal{F}$ (i.e., must be satisfied by every satisfying assignment) and thus it can be added to $\mathcal{F}$ without affecting the set of satisfying assignments. The details on how clauses are learnt are not central to this work. An interested reader may find them in [57].

The search terminates when either a satisfying assignment is found or the instance is determined to be unsatisfiable. Unsatisfiability is determined when a conflict can be derived by unit propagation without making any decisions, i.e., when the empty clause is learned. As the formula $\mathcal{F}$ is then deemed to imply the trivially unsatisfiable empty clause, it follows that $\mathcal{F}$ is unsatisfiable.

2.2 Maximum satisfiability

Maximum satisfiability (MaxSAT) [16, 51] is the optimization version of the Boolean satisfiability problem. It has proven successful as a declarative paradigm for solving optimization problem instances from various domains [16]. A MaxSAT instance $\mathcal{F}_{\texttt{MS}} = \langle \mathcal{F}, w \rangle$ consists of a CNF formula $\mathcal{F}$ and an objective function w over a subset of the variables of $\mathcal{F}$ assigning each variable a natural number as a *weight* (or *cost*). We notate the objective functions as sets of variable–weight pairs, i.e., $w = \{\langle v_1, w_{v_1} \rangle, \langle v_2, w_{v_2} \rangle, \ldots \langle v_n, w_{v_n} \rangle\}$ stands for $w(v_1) = w_{v_1}$, $w(v_2) = w_{v_2}$, …, $w_(v_n) = w_{v_n}$. The variables for which w is defined, are called *objective variables.* We will use $\mathcal{B}(\mathcal{F}_{\texttt{MS}})$ to denote the set of objective variables of a MaxSAT instance $\mathcal{F}_{\texttt{MS}}$. In MaxSAT, given a CNF formula $\mathcal{F}$ and an objective function w, the task is to find a solution τ for $\mathcal{F}$ that minimizes cost defined as

$$\texttt{cost}(\tau, \mathcal{F}_{\texttt{MS}}) = \sum_{v \in \mathcal{B}(\mathcal{F}_{\texttt{MS}})} w(v)\tau(v).$$

In words, the task is to find a solution that minimizes the sum of the weights of the objective variables assigned to 1.

We note that this definition differs from a more traditional view of MaxSAT where the clauses of the formula are divided into *soft* and *hard* clauses and the objective function is defined over the soft clauses. In the more traditional definition of MaxSAT, the task is to find a solution satisfying all hard clauses that minimizes the sum of the weights of the unsatisfied soft clauses. However, any instance corresponding to our definition can be transformed into an instance corresponding to the traditional view of MaxSAT, by introducing a unit soft clause for each negation of an objective variable. A problem instance corresponding to the traditional definition of MaxSAT can be transformed into an instance corresponding to our definition by the so-called *blocking variable transformation* [16], defined as follows. For every soft clause C, introduce to the instance a new (hard) clause $C \lor v_C$ where v_C is a fresh objective variable. The weight of v_C is set to $w(v_C) = w(C)$. The main reason for using our definition instead of the more traditional one is that the algorithms this work focuses on (and actually most modern algorithms for MaxSAT) are simpler to describe using the concept of objective variables.

Sometimes in the literature, a distinction is made between *weighted* and *unweighted* or *partial* and *non-partial* MaxSAT [16, 51]. In weighted MaxSAT, the objective function gives different values to different objective variables, whereas in unweighted MaxSAT all objective variables have equal weight. A partial MaxSAT instance—in the context of the traditional definition—contains soft and hard clauses whereas a non-partial MaxSAT instance contains only soft clauses. To map the concepts of partial and non-partial MaxSAT to our definition of MaxSAT, non-partial MaxSAT instances are instances where each clause is guaranteed to have at least one objective variable in it whereas partial MaxSAT instances are instances without such guarantee. In this work, when we refer to MaxSAT, we refer to the most general definition, i.e., that of *weighted partial MaxSAT*.

Example 2.3

Consider the MaxSAT instance with $\mathcal{F} = (x \lor y) \land (\neg x \lor \neg y)$ and $w = \{\langle x, 1\rangle, \langle y, 2\rangle\}$. $\mathcal{F}$ has two solutions: $\tau_1 = \{\neg x, y\}$ and $\tau_2 = \{x, \neg y\}$. The solution $\tau_1 = \{\neg x, y\}$ has cost 2 since $w(y) = 2$. The solution $\tau_2 = \{x, \neg y\}$ has cost 1 since $w(x) = 1$. This is an optimal solution.

The following example shows how an optimization problem can be encoded as MaxSAT.

Example 2.4

Consider the following optimization problem. There are n possible locations for cellular radio towers and m houses which the network should cover. Each radio tower is associated with a cost. The task is to select a subset of towers that provides the network to every house while minimizing the sum of their costs. The problem can be represented as a graph where towers and houses are nodes and an edge between a tower t_i and a house h_j tells that the tower t_i would provide a network for the house h_j.

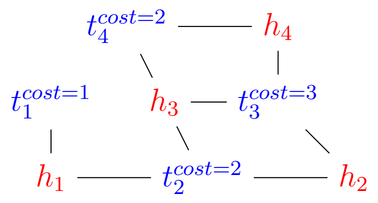


Figure 2.1: An example instance of cellular radio tower optimization problem from Example 2.4.

An example instance of the problem is illustrated in Figure 2.1. There are 4 houses, h_1, h_2, h_3 and h_4 (coloured red) and 4 potential towers to build, t_1, t_2, t_3 and t_4 (coloured blue). Each of the towers is associated with a cost given as $t_i^{cost=c_i}$ in Figure 2.1 implying that building tower t_i costs c_i. Tower t_1 is connected with house h_1 implying it would provide a network to h_1. Tower t_2 is connected with houses h_1, h_2 and h_3, tower t_3 with houses h_2, h_3 and h_4 and tower t_4 with houses h_3 and h_4.
The problem can be encoded to MaxSAT by associating each tower t_i with a Boolean variable t_i and setting the weight of each variable to the cost associated with that tower. The need to cover each house is encoded as clauses. In our example the clauses are $C_{h_1} = (t_1 \lor t_2)$, $C_{h_2} = (t_2 \lor t_3)$, $C_{h_3} = (t_2 \lor t_3 \lor t_4)$ and $C_{h_4} = (t_3 \lor t_4)$. Each of these clauses enforces selecting at least one feasible tower for the corresponding house.
For our example instance shown in the figure 2.1, the encoding gives the MaxSAT instance with $\mathcal{F} = (t_1 \lor t_2) \land (t_2 \lor t_3) \land (t_2 \lor t_3 \lor t_4) \land (t_3 \lor t_4)$, and $w = \{\langle t_1, 1\rangle, \langle t_2, 2\rangle, \langle t_3, 3\rangle, \langle t_4, 2\rangle\}$.
The optimal cost for the MaxSAT instance is 4, achieved by two solutions:
$\tau_1 = \{t_1, \neg t_2, t_3, \neg t_4\}$ that corresponds to building towers t_1 and t_3, and $\tau_2 = \{\neg t_1, t_2, \neg t_3, t_4\}$ that corresponds to building towers t_2 and t_4.

The concept of *(unsatisfiable) cores* is central for many modern algorithms for MaxSAT, especially so for the so-called core-guided algorithms we focus on in this thesis.

Definition 2.2.1. *Given a MaxSAT instance $\mathcal{F}_{\mathtt{MS}} = \langle \mathcal{F}, w\rangle$, a* core *of $\mathcal{F}_{\mathtt{MS}}$ is a subset of objective variables $\kappa \subseteq \mathcal{B}(\mathcal{F}_{\mathtt{MS}})$ for which $\mathcal{F} \land \bigwedge_{v\in\kappa} \neg v$ is unsatisfiable.*

For some intuition on why the concept of core is useful, notice that each core is a subset of objective variables that can not be all assigned to 0 in any satisfying assignment. This means that at least one objective variable in a core must be assigned to 1, thus incurring cost. In this sense, cores provide lower-bound information on the optimal cost of MaxSAT instances.

2.3 Incremental SAT solving

We now overview incremental SAT solving via assumptions and how SAT solvers can be employed to extract cores. As will be explained in Section 2.4, most modern MaxSAT algorithms make extensive use of an incremental SAT solver, in particular, for extracting cores.

In incremental SAT solving [36, 37], a sequence of related instances is solved on one solver that retains its learnt clauses (and possibly other information related, e.g., to the selection of decision variables) on consecutive iterations. This often speeds up search significantly [37].

The central features in modern incremental SAT solvers [37, 57] are the ability to add new clauses to the instance between iterations and the ability to solve under so-called *assumptions*. The set of assumptions is a set of literals defining a partial assignment over the variables of the CNF formula. Given a CNF formula $\mathcal{F}$ and a set of assumptions $\mathcal{A}$, an incremental SAT solver returns a satisfying assignment $\tau \supseteq \mathcal{A}$ if such a τ exists. If such a τ does not exist, the SAT solver returns the set of literals in a conflict clause entailed by $\mathcal{F}$ over the variables in $\mathcal{A}$.

Notice how a conflict clause entailed by $\mathcal{F}$ over $\mathcal{A}$ maps to a concept of core from the MaxSAT point of view (recall Definition 2.2.1). In particular, when a subset of negations of the objective variables is used as a set of assumptions, the conflict clause consists of literals that form a core in terms of MaxSAT.

Example 2.5

Consider the MaxSAT instance with $\mathcal{F} = (\neg x \vee y) \wedge (x \vee \neg y) \wedge (x \vee z)$ and $w = \{\langle x, 1\rangle, \langle y, 1\rangle, \langle z, 1\rangle\}$. Given $\mathcal{A}_1 = \{\neg x, \neg y, \neg z\}$ as the set of assumptions, a SAT solver returns `UNSAT` ("unsatisfiable") on $\mathcal{F}$ under $\mathcal{A}_1$. The SAT solver could return $\kappa_1 = \{x, z\} \subseteq \{\neg l \mid l \in \mathcal{A}_1\}$ as a core, since $\mathcal{F}$ entails this core directly by the clause $(x \vee z)$. The core expresses that $\{\neg x, \neg z\} = \{\neg l \mid l \in \kappa_1\} \subseteq \mathcal{A}_1$ is an inconsistent subset of the assumptions, i.e., $\tau(x) = 1$ or $\tau(y) = 1$ in any solution τ to $\mathcal{F}$. Thus either x or z must incur cost in any solution to $\mathcal{F}$.

2.4 Overview of MaxSAT solving algorithms

We end this chapter with an overview of modern approaches to MaxSAT solving. On a high level, practical algorithms for MaxSAT can be divided to *complete* and *incomplete* algorithms. Incomplete algorithms [42, 4, 30, 49, 22, 47, 35] try to find a good solution (not necessarily an optimal one) in a limited time. In this thesis, we focus on complete algorithms which are guaranteed to find an optimal solution given enough resources. Despite our focus on complete algorithms, the applicability of our contribution is not limited to complete solving since incomplete algorithms in cases employ techniques of complete algorithms [22].

There are four main approaches in modern complete MaxSAT solving: the so-called branch-and-bound [51, 80, 28], model-improving [16], core-guided [16, 39] and implicit hitting set (IHS) approaches [16, 34]. Empirical evaluations have shown that no one approach clearly dominates the others and that the fastest algorithm depends on the instance at hand [17]. We now overview these four main approaches.

Branch-and-bound MaxSAT algorithms [51, 80, 28, 72, 43, 52, 53, 54] explore the space of possible solutions (often presented as a *search tree*) by assigning values to variables one by one in a backtracking style search. During each step, the algorithms check if the current partial assignment could possibly be extended to a new optimal solution; if not, the search backtracks. The check is done by computing a lower bound for the cost of any extension of the current assignment and comparing it to the lowest cost found so far. In different algorithms, the search is optimized by different techniques, e.g., simplifications of the formula, variable selection heuristics, efficient lower bound computation and clause learning [51, 53, 54]. The existing empirical evaluation has shown that even though branch-and-bound often performs well in certain types of instances, e.g., finding largest cuts in random graphs, for large industrial problems the approach generally does not scale well [16, 51]. However, recent results achieved by combining branch-and-bound with clause learning techniques show how branch-and-bound algorithms can still at times be competitive with other modern approaches [53, 54].

Model-improving search for MaxSAT [16, 1, 38, 24] employs a SAT solver incrementally to find better solutions until an optimal solution is found. When solving an instance $\mathcal{F}_{\texttt{MS}} = \langle \mathcal{F}, w \rangle$, model-improving search begins by finding any satisfying assignment for the CNF formula $\mathcal{F}$. Then the search proceeds iteratively towards the optimal cost. If the cost of the current solution is UB, the SAT solver is queried to find a solution which has a cost less than UB, i.e., an assignment that satisfies $\mathcal{F} \wedge \texttt{asCNF}(\sum_{l \in \mathcal{B}(\mathcal{F})} w(l)\tau(l) \leq \text{UB} - 1)$, where `asCNF` denotes a CNF encoding of the inequality. If the instance is satisfiable, a satisfying assignment is extracted, UB is updated to the cost of that assignment and the process is reiterated. When the instance is unsatisfiable, the current UB is the optimal cost, and the last extracted satisfying assignment is one of the optimal solutions to the MaxSAT instance.

Whereas model-improving algorithms proceed from satisfiable SAT instances to an unsatisfiable one, the core-guided approach [16, 39, 66, 59, 58, 55, 8, 7, 10, 64, 65, 69, 26, 2, 45, 71] proceeds in the opposite direction: from unsatisfiable instances to a satisfiable one. The idea is to do a lower-bounding search by iteratively extracting cores of the instance, each core increasing the known lower bound for the optimal cost. More concretely, the search begins by calling a SAT solver while assuming all objective variables to 0, i.e., trying to find a solution with a cost of 0. If the result is unsatisfiable, an unsatisfiable core is extracted and *relaxed*. Relaxing a core transforms the formula in such a way that on the next iteration one of the variables in the core is allowed to incur cost (i.e., be set to 1). When the SAT solver determines the current working instance is satisfiable, the search ends. The satisfying assignment obtained in the last call is then one of the optimal solutions. The approach can be seen as resolving the inconsistencies in the set of objective variables with respect to the formula one by one, each inconsistency increasing the known lower bound for the optimal cost. A more detailed description of the core-guided approach will be given in Chapter 3.

The implicit hitting set (IHS) approach [16, 34, 33, 75] is similar to the core-guided approach in that both extract iteratively cores from SAT solver and proceed from unsatisfiable instances to a final satisfiable instance. The difference is that the IHS approach does not do any transformations on the CNF formula $\mathcal{F}$ (i.e., does not add any clauses). Instead, each SAT solver call is made on original $\mathcal{F}$ under different sets of assumptions over $\mathcal{B}(\mathcal{F})$. To select a set of assumptions for the next SAT solver call, a minimum-cost hitting set is computed over the found cores. A minimum-cost hitting set represents an optimal way of resolving the cores found so far. The objective variables in a minimum-cost hitting set will be allowed to incur cost on the next SAT solver call. The algorithm terminates when the SAT solver determines that the instance is satisfiable at which point the found solution is guaranteed to be an optimal one.

3 Core-guided MaxSAT solving with soft cardinality constraints

In this chapter, we give a more detailed description of modern core-guided MaxSAT algorithms. We focus, in particular, on OLL [65, 3] and PMRES [69] as the two algorithms in the context of which we will also detail our structure-sharing technique later in Chapter 4.

3.1 Overview

Algorithm 1 provides a generic abstraction of the core-guided approach in pseudocode. When invoked on a MaxSAT instance $\mathcal{F}_{\texttt{MS}}$, the search begins by initializing a working instance $\mathcal{F}^1_{\texttt{MS}}$ to $\mathcal{F}_{\texttt{MS}}$ (Line 1) and the known lower-bound for cost LB to 0 (Line 2). In each iteration of the main search loop (Lines 3-7), a core of the current working instance $\mathcal{F}^i_{\texttt{MS}}$ is extracted (Line 4). The use of an incremental SAT solver is abstracted into the function Extract-Core that takes as input the current CNF formula $\mathcal{F}^i$ and the objective variables $\mathcal{B}(\mathcal{F}^i_{\texttt{MS}})$ of the current working instance to use their negations as assumptions. The function returns a triple (res, κ, τ). If res $=$ `SAT`, then τ is an assignment satisfying $\mathcal{F}^i$ that sets $\tau(v) = 0$ for each $v \in \mathcal{B}(\mathcal{F}^i_{\texttt{MS}})$. Such a τ has $\texttt{cost}(\mathcal{F}^i_{\texttt{MS}}, \tau) = 0$ and will be optimal for both $\mathcal{F}^i_{\texttt{MS}}$ and $\mathcal{F}_{\texttt{MS}}$. In this case, Algorithm 1 terminates and returns τ (Line 5).

Algorithm 1: CG a generic view on core-guided MaxSAT solving.

```
Input: A MaxSAT instance $\mathcal{F}_{\texttt{MS}} = \langle \mathcal{F}, w \rangle$
Output: An optimal solution $\tau$ to $\mathcal{F}_{\texttt{MS}}$
1 $\mathcal{F}^1_{\texttt{MS}} \leftarrow \mathcal{F}_{\texttt{MS}}$
2 $\text{LB} \leftarrow 0$
3 for $i = 1, \ldots$ do
4     $(res, \kappa, \tau) \leftarrow \textsc{Extract-Core}(\mathcal{F}^i, \mathcal{B}(\mathcal{F}^i_{\texttt{MS}}))$
5     if $res$=SAT then return $\tau$
6     $\text{LB} \leftarrow \text{LB} + \texttt{minw}(\kappa)$
7     $\mathcal{F}^{i+1}_{\texttt{MS}} \leftarrow \textsc{Relax}(\mathcal{F}^i_{\texttt{MS}}, \kappa)$
```

If res $= \texttt{UNSAT}$ instead, then κ is a core of $\mathcal{F}^i_{\texttt{MS}}$ expressed over the variables in $\mathcal{B}(\mathcal{F}^i_{\texttt{MS}})$. Recall that—by definition—a core must have at least one objective variable in it set to 1 in any solution to $\mathcal{F}_{\texttt{MS}}$, thus incurring cost. A core will incur at least $\texttt{minw}(\kappa) = \min(\{w(v) \mid v \in \kappa\})$ cost. The current lower-bound LB is increased by $\texttt{minw}(\kappa)$ (Line 6). The core κ is then relaxed by function Relax (Line 7). Relaxing a core allows, in a controlled way, one of its variables to be set to 1 in subsequent iterations.

The main idea of core relaxation is perhaps easiest to understand when considering the case where each variable in the core κ has the same weight. In such a case, core relaxation results in the following two things.

1. Relaxing a core enables assigning variables of κ to 1 by setting their weights to 0. This efficiently removes the negations of the variables of κ from the set of assumptions in the subsequent SAT solver calls.

2. Relaxing a core restricts the number of variables of κ assigned to 1 by introducing a *cardinality constraint* over the variables. A cardinality constraint over a core κ enforces a restriction to the sum $\sum_{v \in \kappa} \tau(v)$ to some bound k in any solution τ. Initially, $\sum_{v \in \kappa} \tau(v) \leq 1$ is enforced.

Notice that even though at first the cardinality constraint limits the sum $\sum_{v \in \kappa} \tau(v)$ to at most 1, it might be that in all optimal solutions more than one variable of a core κ is set to 1. This is the reason why cardinality constraints are introduced in a way that enables later relaxing them further. Core-guided MaxSAT algorithms differ mainly in how the cardinality constraints are constructed. Roughly speaking, the earliest core-guided algorithms added hard constraints to the formula when relaxing a core [39, 59, 58, 55, 8, 7], whereas modern ones tend to add so-called soft cardinality constraints, enabling efficient use of incremental SAT solvers [10, 69, 65, 2, 62].

Relaxing a core κ in the more general weighted case is detailed in pseudocode in Algorithm 2. The difference to the unweighted case is that the weights of the variables in the core are reduced by $\texttt{minw}(\kappa) = \min(\{w(v) \mid v \in \kappa\})$ (Lines 1-4) rather than setting them to 0. This corresponds to the so-called clause cloning technique [8, 9, 26, 65, 69]. The main idea of clause cloning is to "split" each variable v in κ with $w(v) > \texttt{minw}(\kappa)$ into two variables: v^1 with weight $w(v^1) = \texttt{minw}(\kappa)$ and v^2 with weight $w(v^2) = w(v) - \texttt{minw}(\kappa)$[1]. Variable v in κ is replaced with variable v^1. When this is done for every $v \in \kappa$, κ is relaxed with all variables having weight equal to $\texttt{minw}(\kappa)$ as in the unweighted case.

[1]For details, see for example [16]

Algorithm 2: Relax in core-guided algorithm with soft cardinality constraints.

```
Input: A MaxSAT instance $\mathcal{F}_{\texttt{MS}} = \langle \mathcal{F}, w \rangle$ and a core $\kappa$
Output: A transformed instance $\mathcal{F}'_{\texttt{MS}} = \langle \mathcal{F}', w' \rangle$ where the core is relaxed
1 $w^\kappa \leftarrow \texttt{minw}(\kappa)$
2 $w' \leftarrow w$
3 for $v \in \kappa$ do
4     $w'(v) \leftarrow w(v) - w^\kappa$
5 $\mathcal{F}_{\text{CC}}, w_{\text{CC}} \leftarrow$ Build-Card-Constraint$(\kappa, w^\kappa)$
6 return $\langle \mathcal{F} \wedge \mathcal{F}_{\text{CC}}, w' \cup w_{\text{CC}} \rangle$
```

However, the method of reducing weights has the same effect as clause cloning without introducing duplicate clauses (or variables) to the instance [2, 23]. After the weights are reduced, a cardinality constraint is constructed over the variables in the core in function Build-Card-Constraint (Line 5). Build-Card-Constraint returns the cardinality constraint encoded in CNF ($\mathcal{F}_{\text{CC}}$) and the weights of the fresh objective variables introduced by the constraint (w_{CC}). $\mathcal{F}_{\text{CC}}$ and w_{CC} are introduced to the working instance (Line 6). We will detail concrete ways of encoding cardinality constraints in Sections 3.2.1, 3.3 and 4.2.1.

The algorithms focused on in this thesis make use of the soft cardinality constraints, the underlying idea of which is that constraints are associated with variables whose values are enforced to correspond to whether an assignment of values satisfies the constraint or not. This enables the use of the assumption interface of a SAT solver to enforce or not enforce the constraints. In practice, the encodings determine a set of *output variables* for a given set of *input variables*. Conceptually, the input variables (variables in the core) determine the values of the output variables. Enforcing values to the output variables via the assumption interface will restrict the assignments of the input variables, i.e., the assignments of the variables of the core. This will become more concrete when we represent the actual algorithms and the cardinality constraint encodings they use.

3.2 OLL

The first of the two algorithms we will extend with structure-sharing is OLL [65, 3]. OLL was originally proposed in the context of answer set programming [3] and later adapted to MaxSAT [65].

Core relaxation in OLL works as follows. After reducing the weights of core κ, OLL introduces a set of cardinality constraints to the instance. Each cardinality constraint has a corresponding output variable that is enforced to 1 if the constraint holds on a given assignment and to 0 if it does not. To be more precise, when relaxing a core κ, OLL adds to the working instance

$$o_i^\kappa \leftrightarrow \sum_{v \in \kappa} v \geq i \text{ for } i = 2, 3, ..., |\kappa|$$

as constraints. The variables o_i^κ are the output variables of these constraints.

A helpful way of thinking about the output variables of OLL is that defining variables $\{o_1^\kappa,\, o_2^\kappa, \ldots o_{|\kappa|}^\kappa\}$ for κ with the semantics $o_i^\kappa \leftrightarrow (\sum_{v \in \kappa} v \geq i)$ corresponds to *sorting* the values in κ. If a solution sets k of the variables in κ to 1 and $|\kappa| - k$ to 0, it also sets $o_1^\kappa = 1, o_2^\kappa = 1, \ldots, o_k^\kappa = 1$ and $o_{k+1}^\kappa = 0, o_{k+2}^\kappa = 0, \ldots, o_{|\kappa|}^\kappa = 0$. Notice that the output variable $o_1^\kappa \leftrightarrow \sum_{v \in \kappa} v \geq 1$ is not needed in the context of OLL, since it would limit the number of variables assigned 1 in a core to 0. By the definition of core, any satisfying assignment must set at least one variable in the core to 1, implying $o_1^\kappa = 1$ in any solution to the formula.

The output variables o_i^κ $(i \geq 2)$ with weight $\texttt{minw}(\kappa)$ will become new objective variables in the working instance. In the next iteration(s) the output variables will be assumed to 0, enforcing the constraints that limit the number of the variables assigned 1 in the core. Later in the search, the output variables may also participate in cores in which case they will be handled like any other objective variables: their weights are decreased and new cardinality constraints are constructed over them as for any other objective variable.

The following example gives an example execution of the OLL algorithm.

Example 3.1

Consider the following MaxSAT instance:

$$\begin{aligned} \mathcal{F}^1 = &\quad (x \vee y \vee z) \wedge (x \vee y) \wedge (x \vee z) \wedge (y \vee z), \\ w^1 = &\quad \{\langle x, 4\rangle, \langle y, 5\rangle, \langle z, 2\rangle\}. \end{aligned}$$

Assume that the SAT solver, invoked on $\mathcal{F}^1$ under the set of assumptions $\mathcal{A}^1 = \{\neg x, \neg y, \neg z\}$, returns as the first core $\kappa_1 = \{x, y, z\}$. The weights of x, y and z are decreased by $\texttt{minw}(\kappa_1) = 2$. The new weights are $w(x) = 2$, $w(y) = 3$ and $w(z) = 0$. z will be removed from the assumptions in the next iterations since its weight is 0. New output variables and their cardinality constraints are $o_2^{\kappa_1} \leftrightarrow (x + y + z \geq 2)$ and $o_3^{\kappa_1} \leftrightarrow (x + y + z \geq 3)$. Both $o_2^{\kappa_1}$ and $o_3^{\kappa_1}$ are initialized with weight $\texttt{minw}(\kappa_1) = 2$.

The lower bound increases to $\mathrm{LB}^2 = \mathtt{minw}(\kappa_1) = 2$.
In the next iteration, the state of the algorithm is

$$\begin{aligned}
\mathcal{F}^2 = \quad & (x \vee y \vee z) \wedge (x \vee y) \wedge (x \vee z) \wedge (y \vee z) \\
& \wedge \mathtt{asCNF}(o_2^{\kappa_1} \leftrightarrow (x + y + z \geq 2)) \\
& \wedge \mathtt{asCNF}(o_3^{\kappa_1} \leftrightarrow (x + y + z \geq 3)), \\
w^2 = \quad & \{\langle x, 2\rangle, \langle y, 3\rangle, \langle o_2^{\kappa_1}, 2\rangle, \langle o_3^{\kappa_1}, 2\rangle\}, \\
\mathcal{A}^2 = \quad & \{\neg x, \neg y, \neg o_2^{\kappa_1}, \neg o_3^{\kappa_1}\}, \\
\mathrm{LB}^2 = \quad & 2.
\end{aligned}$$

Assume that the next core returned by the SAT solver on $\mathcal{F}^2$ under $\mathcal{A}^2$ is $\kappa_2 = \{o_2^{\kappa_1}\}$. The weight of variable $o_2^{\kappa_1}$ is set to 0 and thus $\neg o_2^{\kappa_1}$ is removed from the assumptions. Notice that cardinality constraints are not needed for κ_2 since $|\kappa_2| = 1$, i.e., the number of variables assigned to 1 is already trivially limited to 1. The lower bound increases by $\mathtt{minw}(\kappa_2) = 2$ to $\mathrm{LB}^3 = 4$. The state of the algorithm is in the next iteration

$$\begin{aligned}
\mathcal{F}^3 = \quad & (x \vee y \vee z) \wedge (x \vee y) \wedge (x \vee z) \wedge (y \vee z) \\
& \wedge \mathtt{asCNF}(o_2^{\kappa_1} \leftrightarrow (x + y + z \geq 2)) \\
& \wedge \mathtt{asCNF}(o_3^{\kappa_1} \leftrightarrow (x + y + z \geq 3)), \\
w^3 = \quad & \{\langle x, 2\rangle, \langle y, 3\rangle, \langle o_3^{\kappa_1}, 2\rangle\}, \\
\mathcal{A}^3 = \quad & \{\neg x, \neg y, \neg o_3^{\kappa_1}\}, \\
\mathrm{LB}^3 = \quad & 4.
\end{aligned}$$

Assume that the next core returned by the SAT solver on $\mathcal{F}^3$ under $\mathcal{A}^3$ is $\kappa_3 = \{x, y\}$. After relaxing this core, the state of the algorithm is

$$\begin{aligned}
\mathcal{F}^4 = \quad & (x \vee y \vee z) \wedge (x \vee y) \wedge (x \vee z) \wedge (y \vee z) \\
& \wedge \mathtt{asCNF}(o_2^{\kappa_1} \leftrightarrow (x + y + z \geq 2)) \\
& \wedge \mathtt{asCNF}(o_3^{\kappa_1} \leftrightarrow (x + y + z \geq 3)) \\
& \wedge \mathtt{asCNF}(o_2^{\kappa_3} \leftrightarrow (x + y \geq 2)), \\
w^4 = \quad & \{\langle y, 1\rangle, \langle o_3^{\kappa_1}, 2\rangle, \langle o_2^{\kappa_3}, 2\rangle\}, \\
\mathcal{A}^4 = \quad & \{\neg y, \neg o_3^{\kappa_1}, \neg o_2^{\kappa_3}\}, \\
\mathrm{LB}^4 = \quad & 6.
\end{aligned}$$

The instance $\mathcal{F}^4$ is satisfiable under $\mathcal{A}^4$, by $\tau = \{x, \neg y, z\}$. The optimal cost is 6 since it is the final lower bound LB^4.
Notice that the values of the input variables of cardinality constraints define the

values of all the other variables[1]. Because of this, when presenting solutions we give values for the variables of the original formula only.

Next, we sketch a proof of the correctness of OLL. This follows from the cardinality constraints of OLL having the following so-called counting property.

Definition 3.2.1. *A cardinality constraint encoding has a* counting property *for κ input variables if the encoding defines a set of output variables in such a way, that any satisfying assignment τ assigning exactly k input variables to* 1 *assigns exactly $k-1$ output variables to* 1.

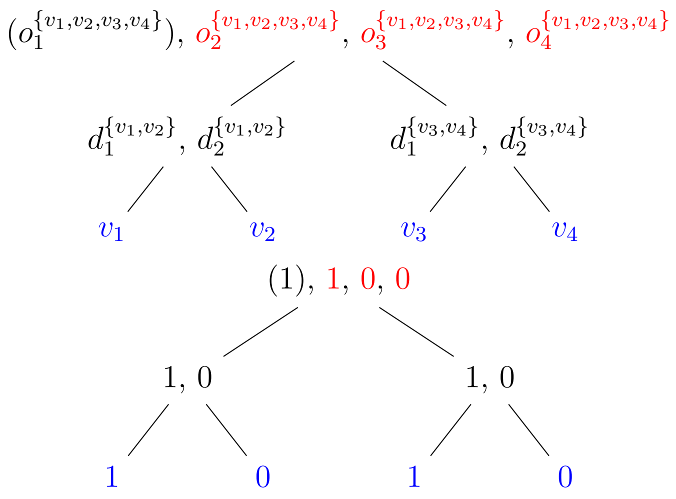
As the cardinality constraints of OLL have the counting property, the correctness of OLL is based on the following proposition.

Proposition 3.2.1. *Assume that a core-guided algorithm as defined in Algorithms 1 and 2 has found and relaxed $i-1$ cores and has found the ith core κ_i. Let $\mathcal{F}^i_{\texttt{MS}}$ be the instance before relaxing κ_i and $\mathcal{F}^{i+1}_{\texttt{MS}}$ the instance after relaxation. Assume that the used cardinality constraint encoding has the counting property (Definition 3.2.1) and the new output variables are each initialized with weight* $\texttt{minw}(\kappa_i)$. *Then for any solution τ to $\mathcal{F}^i_{\texttt{MS}}$,* $\texttt{cost}(\tau', \mathcal{F}^{i+1}_{\texttt{MS}}) = \texttt{cost}(\tau, \mathcal{F}^i_{\texttt{MS}}) - \texttt{minw}(\kappa_i)$ *where τ' is assignment τ completed to satisfy the new cardinality constraints in $\mathcal{F}^{i+1}_{\texttt{MS}}$.*

Proof. Let k be the number of variables the assignment τ assigns to 1 in the core κ_i. Because the weights of the variables in the core are reduced by $\texttt{minw}(\kappa_i)$ in core relaxation, the sum of their weights will be $k \cdot \texttt{minw}(\kappa_i)$ less in $\mathcal{F}^{i+1}_{\texttt{MS}}$ than in $\mathcal{F}^i_{\texttt{MS}}$. The new output variables in $\mathcal{F}^{i+1}_{\texttt{MS}}$ each have weight $\texttt{minw}(\kappa_i)$, and exactly $k-1$ of these will be assigned to 1 by any satisfying assignment setting k of the variables in κ_i to 1. This adds $(k-1)\texttt{minw}(\kappa_i)$ to $\texttt{cost}(\tau', \mathcal{F}^{i+1}_{\texttt{MS}})$. All other objective variables of $\mathcal{F}^i_{\texttt{MS}}$ and $\mathcal{F}^{i+1}_{\texttt{MS}}$ have equal weights. Thus $\texttt{cost}(\tau', \mathcal{F}^{i+1}_{\texttt{MS}}) = \texttt{cost}(\tau, \mathcal{F}^i_{\texttt{MS}}) - k \cdot \texttt{minw}(\kappa_i) + (k-1)\texttt{minw}(\kappa_i) = \texttt{cost}(\tau, \mathcal{F}^i_{\texttt{MS}}) - \texttt{minw}(\kappa_i)$. □

There are various alternative ways to encode the cardinality constraints of OLL into CNF. In this thesis, we make use of the totalizer encoding [20]. Although our structure-sharing technique is not specific for totalizers, the applicability of structure-sharing to each of the different cardinality constraint encodings would need to be studied separately in each case and is beyond the scope of this thesis.

[1]Strictly speaking, the input variables define the values of all other variables when equivalence semantics is used in the encoding (see Section 4.2.1). However, this detail does not affect the correctness of our approach.



$(o_1^{\{v_1,v_2,v_3,v_4\}})$, $o_2^{\{v_1,v_2,v_3,v_4\}}$, $o_3^{\{v_1,v_2,v_3,v_4\}}$, $o_4^{\{v_1,v_2,v_3,v_4\}}$
$d_1^{\{v_1,v_2\}}, d_2^{\{v_1,v_2\}}$
$d_1^{\{v_3,v_4\}}, d_2^{\{v_3,v_4\}}$
v_1
v_2
v_3
v_4
(1), 1, 0, 0
1, 0
1, 0
1
0
1
0

Figure 3.1: An example of a totalizer, its variables (top) and a feasible assignment of values (bottom).

3.2.1 The totalizer encoding of cardinality constraints

A totalizer [20] is a CNF encoding of cardinality constraints that can be employed to encode the cardinality constraints of OLL. Given a set κ of input variables, the totalizer encoding produces a CNF formula that defines $|\kappa|$ output variables o_i^κ with semantics

$$o_i^\kappa \leftrightarrow \sum_{v \in \kappa} v \geq i.$$

The structure of the encoding can be viewed as a binary tree. An example of a totalizer constructed over $\{v_1, v_2, v_3, v_4\}$ is illustrated in Figure 3.1. The input variables occur as leaves (variables v_i coloured blue in Figure 3.1). Every internal node N is associated with as many variables as there are leaves in the subtree rooted at N. In Figure 3.1, these variables of N are labelled as d_i^S, where S is the set of input variables in the subtree rooted at N, and i the index of the variable in the N. The values of these variables are enforced to correspond to the values of the variables in set S, but in sorted order, 1s before 0s. In other words, the semantics of the variables is defined as

$$d_i^S \leftrightarrow \sum_{v \in S} v \geq i.$$

The values of the variables in the root correspond to the values of all input variables in sorted order; they are the output variables. In Figure 3.1 the variables in the root are denoted with the letter o (as output variables). In the context of OLL, the variables $2, 3, \ldots, |\kappa|$ in the root of a totalizer are the output variables that are of interest. In Figure 3.1 these relevant output variables are coloured red. The redundant first output variable is in parenthesis.

Figure 3.1 also illustrates an example of a feasible assignment of values to the variables in the totalizer, i.e., an assignment that follows the semantics of the variables. From the point of view that the totalizer is a CNF formula, such an assignment satisfies the formula. Notice how the values of the variables of each node are in sorted order, and the values correspond to the values of the variables in the leaves of the subtree.

For $n = |\kappa|$ input variables, a totalizer structured as a balanced binary tree introduces $\Theta(n \log n)$ variables and $\Theta(n^2)$ clauses. The number of variables follows from the fact that each inner node of the totalizer introduces $|L| + |R|$ variables, where $|L|$ is the number of input variables in the left subtree of the node and $|R|$ correspondingly in the right subtree. On each $\Theta(\log n)$ levels of the tree, there are $\Theta(n)$ variables, hence the limit $\Theta(n \log n)$. The number of clauses follows from the fact that the CNF encoding introduces $\Theta(|L||R|)$ clauses for each node (the clauses will be discussed in Section 4.2.1). The total number of clauses will be $\Theta((\frac{n}{2})^2 + 2(\frac{n}{4})^2 + 4(\frac{n}{8})^2 + \cdots + n) = \Theta(n^2)$. However, if the totalizer tree is not balanced, the number of variables is bounded by $\Theta(n^2)$. The number of clauses in the encoding is still bounded by $\Theta(n^2)$ since each pair of input variables is sorted by $\Theta(1)$ clauses.

The bound $\Theta(n^2)$ can be considered relatively large. However, in practice, the encodings tend not to become large since totalizers can be constructed incrementally [64, 63, 62]. The incremental encoding makes use of the fact that to enforce bound $\sum_{v \in \kappa} \tau(v) \leq k$ it is sufficient to construct the totalizer only partially. A (balanced) incremental totalizer that enforces bound k introduces $\Theta(n \log k)$ variables and $\Theta(nk)$ clauses.[1] Enabling a totalizer to enforce larger bounds can be done by introducing the clauses and variables of the totalizer needed for the larger bound that are not present in the current partial totalizer. Notice, that enforcing a small bound k enforces implicitly also all bounds $k' > k$. At first, bound $k = 2$ is enforced, and larger bounds are only enforced on demand.

[1]With n input variables and bound k, an incremental totalizer has $\Theta(\log n - \log k)$ levels that introduce total $\Theta(k + 2k + 4k + \cdots + n) = \Theta(n)$ variables. The other $\Theta(\log k)$ levels introduce $\Theta(n)$ variables each, totalling $\Theta(n \log k)$ variables. The number of variables is thus bounded by $\Theta(n \log k)$.

The $\Theta(\log n - \log k)$ levels introduce total $\Theta(k^2 + 2k^2 + 4k^2 + \cdots + 2^{\log n - \log k} k^2) = \Theta(2^{\log n - \log k} k^2) = \Theta(\frac{n}{k} k^2) = \Theta(nk)$ clauses. The other $\Theta(\log k)$ levels introduce total $\Theta(\frac{n}{k} k^2 + 2\frac{n}{k}(\frac{k}{2})^2 + 4\frac{n}{k}(\frac{k}{4})^2 + \cdots + n) = \Theta(nk)$ clauses. The total number of clauses is thus bounded by $\Theta(nk)$.

3.3 PMRES

The second algorithm we extend with structure-sharing is PMRES [69]. Like OLL, PMRES employs soft cardinality constraints. However, the output variables PMRES introduces are semantically different from the ones introduced by OLL. Given a core $\kappa = \{v_1, v_2, \ldots, v_{|\kappa|}\}$, PMRES adds one at-most-one-constraint (i.e., $\sum_{v\in\kappa} v \leq 1$), enforced by $|\kappa| - 1$ output variables o_i. Each output variable gets weight $\texttt{minw}(\kappa)$. In the original algorithm [69], the semantics of the output variables were

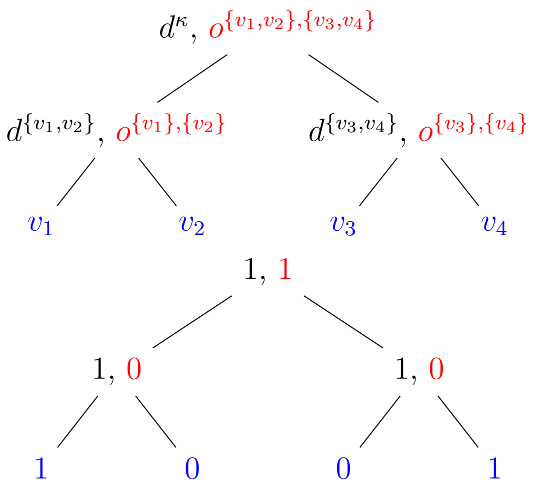
$$o_i \leftrightarrow \left(\bigvee_{j\leq i} v_j \wedge v_{i+1}\right) \text{ for } 1 \leq i \leq |\kappa| - 1.$$

However, in this thesis, we will consider a generalization of PMRES, which we call *generalized PMRES*. The difference to the original version is that generalized PMRES enables more ways of constructing the cardinality constraint for a given set of input variables κ. The main reason for focusing on the generalization is that in some cases it allows for more structure-sharing as will be explained in Section 4.1.1.

We denote the generalized PMRES cardinality constraint encoding we use by GPMRES-ENC. For a set of input variables κ, GPMRES-ENC(κ) defines $|\kappa| - 1$ output variables with semantics

$$o^{S_1,S_2} \leftrightarrow \left(\bigvee_{v_i\in S_1} v_i \wedge \bigvee_{v_j\in S_2} v_j\right), \text{ where } S_1 \subseteq \kappa \text{ and } S_2 \subseteq \kappa. \tag{3.1}$$

The sets S_1 and S_2 are selected for each output variable in such a way that the cardinality constraint encoding has the counting property (recall Definition 3.2.1). Then the correctness can be established using Proposition 3.2.1. In practice, the selection of S_1 and S_2 is based on viewing the structure as a binary tree. Figure 3.2 illustrates an example of the structure over $\kappa = \{v_1, v_2, v_3, v_4\}$. The input variables occur as leaves (variables v_i coloured blue in Figure 3.2). Each inner node N of the binary tree structure is associated with two variables, $d^{L\cup R}$ and $o^{L,R}$, where L is the set of input variables of the left subtree of N and R is the set of the input variables of the right subtree of N. The variables $d^{L\cup R}$ are auxiliary variables and the variables $o^{L,R}$ are output variables (coloured red in Figure 3.2). Variable $d^{L\cup R}$ will be assigned to 1 by a satisfying assignment τ if and only if at least one input variable in the corresponding subtree is assigned to 1. Efficiently $d^{L\cup R} \leftrightarrow \bigvee_{v_i\in L\cup R} v_i$. The variable $o^{L,R}$ will be assigned to 1 if and only if both subtrees



d^κ, $o^{\{v_1,v_2\},\{v_3,v_4\}}$
$d^{\{v_1,v_2\}}$, $o^{\{v_1\},\{v_2\}}$
$d^{\{v_3,v_4\}}$, $o^{\{v_3\},\{v_4\}}$
v_1
v_2
v_3
v_4
1, 1
1, 0
1, 0
1
0
0
1

Figure 3.2: Variables of a generalized PMRES cardinality constraint over $\kappa = v_1, v_2, v_3, v_4$ structured as a balanced tree (top), and an example of a feasible assignment of values (bottom). Two input variables (blue) being assigned to 1 enforces assigning one of the output variables (red) to 1.

have at least one input variable assigned to 1. Efficiently, the semantics of the output variables are

$$o^{L,R} \leftrightarrow \Big(\bigvee_{v_i \in L} v_i \wedge \bigvee_{v_j \in R} v_j \Big).$$

The sets S_1 and S_2 of each output variable o^{S_1,S_2} thus consist of the input variables of the leaves of the left and right subtree of the node into which that output variable belongs. Figure 3.2 also illustrates an example of a feasible assignment of values on GPMRES-ENC. Two input variables assigned to 1 result in one output variable being assigned to 1.

To see that GPMRES-ENC has the counting property, notice that in the binary tree of GPMRES-ENC, assigning a set of inputs to 1 enforces the output variables on their pairwise lowest common ancestors to 1. In any binary tree, a subset of input variables of cardinality k has a set of pairwise lowest common ancestors of cardinality $k-1$. Thus enforcing k input variables to 1, enforces $k-1$ output variables to 1. Based on this observation, the correctness of the algorithm can be proved using Proposition 3.2.1 we presented in the context of OLL as the proof of Proposition 3.2.1 essentially relies on the counting property. In Appendix A, we formally establish that any way of selecting the sets S_1 and S_2 for each output variable as defined in Equation 3.1 in a way that satisfies the counting property can be viewed as a binary tree.

Notice that the cardinality constraints of the original PMRES algorithm can be seen as

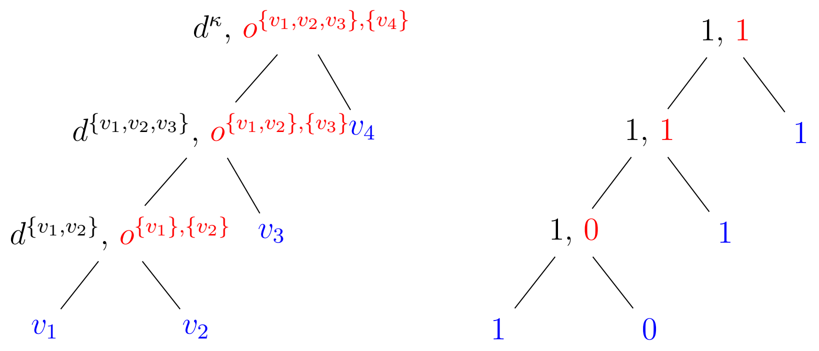

d^{κ}, $o^{\{v_1,v_2,v_3\},\{v_4\}}$
$d^{\{v_1,v_2,v_3\}}$, $o^{\{v_1,v_2\},\{v_3\}}$
v_4
$d^{\{v_1,v_2\}}$, $o^{\{v_1\},\{v_2\}}$
v_3
v_1
v_2
1, 1
1, 1
1
1, 0
1
1
0

Figure 3.3: A chain-like PMRES cardinality constraint over $\kappa = \{v_1, v_2, v_3, v_4\}$ corresponding to the original PMRES algorithm (left) and an example of a feasible assignment of values to the variables (right). Three of the input variables (blue) being assigned to 1 enforces assigning two of the output variables (red) to 1.

chain-like binary trees and as such as a special case of GPMRES-ENC. An example of a binary tree corresponding to a cardinality constraint of the original PMRES is illustrated in Figure 3.3. Figure 3.3 also illustrates an example of a feasible assignment of values in the structure.

For $n = |\kappa|$ input variables, the GPMRES-ENC(κ) will contain $\Theta(n)$ variables and $\Theta(n)$ clauses. Compared to totalizers, the size is smaller. (Recall that a totalizer has $\Theta(n \log n)$ variables and $\Theta(n^2)$ clauses.) However, in practice, the differences in the sizes of the encodings are not this large, since totalizers can be constructed incrementally (recall Section 3.2.1). In PMRES, the structure of the selected tree does not affect the size of the cardinality constraint encoding. Both the balanced tree of Figure 3.2 and the chain-like of in Figure 3.3 introduce as many clauses and variables.

3.4 Weight-aware core extraction

We end this chapter by describing the weight-aware core extraction technique (WCE) [23] which is essential for the efficient realization of our structure-sharing technique (as will be explained later in Chapter 4).

WCE is a technique for extracting multiple cores during each iteration, and it can be incorporated into any core-guided algorithms we are aware of, and even into IHS algorithms [23, 78, 76]. Here our description focuses on WCE in the context of core-guided algorithms.

A core-guided algorithm with soft cardinality constraints extended with WCE is repre-

Algorithm 3: CG+WCE a core-guided MaxSAT algorithm with soft cardinality constraints extended with WCE.

```
Input: A MaxSAT instance $\mathcal{F}_{\texttt{MS}} = \langle \mathcal{F}, w \rangle$
Output: An optimal solution $\tau$ to $\mathcal{F}_{\texttt{MS}}$
1  $\mathcal{F}^1_{\texttt{MS}} \leftarrow \mathcal{F}_{\texttt{MS}}$
2  $\mathcal{K} \leftarrow \emptyset$
3  for $i = 1, \ldots$ do
4      $(res, \kappa, \tau) \leftarrow$ Extract-Core$(\mathcal{F}^i, \mathcal{B}(\mathcal{F}^i_{\texttt{MS}}))$
5      if $res$=SAT and $|\mathcal{K}| = 0$ then return $\tau$
6      else if $res$=SAT and $|\mathcal{K}| > 0$ then
7          $\mathcal{F}^{i+1}_{\texttt{MS}} \leftarrow$ Build-Card-Constraints$(\mathcal{F}^i_{\texttt{MS}}, \mathcal{K})$
8          $\mathcal{K} \leftarrow \emptyset$
9      else
10         $w^\kappa \leftarrow \texttt{minw}(\kappa)$
11         for $v \in \kappa$ do
12             $w(v) \leftarrow w(v) - w^\kappa$
13         $\mathcal{K}.add(\langle \kappa, w^\kappa \rangle)$
```

sented in pseudocode as Algorithm 3. The algorithm first initializes the working instance (Line 1) and a set $\mathcal{K}$ as an empty set (Line 2). The set $\mathcal{K}$ contains the cores that are found during the current iteration. When a core κ is found, instead of continuing immediately with core relaxation, only the weights of the variables of κ are decreased, and κ and its weight $\texttt{minw}(\kappa)$ are added to the set $\mathcal{K}$ (Lines 9–13). Then the algorithm continues finding more cores with the updated set of assumptions. Only when no more cores are found, the cardinality constraints for the found cores are constructed and the set $\mathcal{K}$ is emptied (Lines 6–8). The core-guided algorithm extended with WCE terminates when res = SAT and the set $\mathcal{K}$ is already empty, i.e., cardinality constraints are constructed for all found cores (Line 5).

An important observation for the structure-sharing we will detail in Chapter 4 is that instead of constructing cardinality constraints for one core at a time, with WCE a *set* of cardinality constraints is constructed for a *set* of cores at a time.

4 Structure-sharing

In this chapter, we detail the main technical contribution of this work, namely, the structure-sharing technique for core-guided algorithms using soft cardinality constraints. We first introduce the structure-sharing technique and discuss how it can be employed in the context of OLL and PMRES. The main aim is to decrease the number of variables and clauses in the cardinality constraint structures. Then we continue to show how structure-sharing can enable additional propagation which may speed up search.

4.1 Structure-sharing

To motivate structure-sharing, consider the following example of a situation where a core-guided algorithm with WCE ends up constructing cardinality constraints for two overlapping cores.

Example 4.1

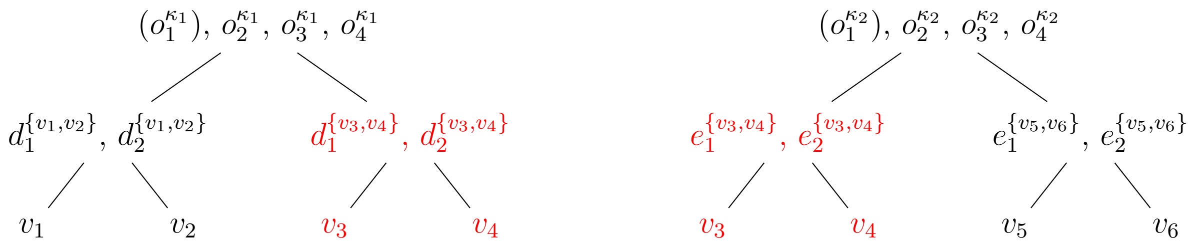
Consider the following MaxSAT instance:

$$\begin{aligned} \mathcal{F}^1 = &\quad (v_1 \lor v_2 \lor v_3 \lor v_4) \land (v_3 \lor v_4 \lor v_5 \lor v_6), \\ w^1 = &\quad \{\langle v_1, 1\rangle, \langle v_2, 1\rangle, \langle v_3, 2\rangle, \langle v_4, 2\rangle, \langle v_5, 1\rangle, \langle v_6, 1\rangle\}. \end{aligned}$$

Assume that a core-guided algorithm extended with WCE extracts as the first core $\kappa_1 = \{v_1, v_2, v_3, v_4\}$. The weights are decreased, the LB is increased and then the search continues within the WCE loop. The state of the algorithm on the next SAT solver call is

$$\begin{aligned} \mathcal{F}^2 = &\quad (v_1 \lor v_2 \lor v_3 \lor v_4) \land (v_3 \lor v_4 \lor v_5 \lor v_6), \\ w^2 = &\quad \{\langle v_3, 1\rangle, \langle v_4, 1\rangle, \langle v_5, 1\rangle, \langle v_6, 1\rangle\}, \\ \mathcal{A}^2 = &\quad \{\neg v_3, \neg v_4, \neg v_5, \neg v_6\}, \\ \mathrm{LB}^2 = &\quad 1. \end{aligned}$$

Assume that the algorithm extracts as the next core $\kappa_2 = \{v_3, v_4, v_5, v_6\}$. After the weights are updated, every weight is zero, and with an empty set of assumptions, the instance is satisfiable. The next step would be to construct cardinality constraints over the extracted cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$ and $\kappa_2 = \{v_3, v_4, v_5, v_6\}$.



$(o_1^{\kappa_1}), o_2^{\kappa_1}, o_3^{\kappa_1}, o_4^{\kappa_1}$
$d_1^{\{v_1,v_2\}}, d_2^{\{v_1,v_2\}}$
$d_1^{\{v_3,v_4\}}, d_2^{\{v_3,v_4\}}$
v_1
v_2
v_3
v_4
$(o_1^{\kappa_2}), o_2^{\kappa_2}, o_3^{\kappa_2}, o_4^{\kappa_2}$
$e_1^{\{v_3,v_4\}}, e_2^{\{v_3,v_4\}}$
$e_1^{\{v_5,v_6\}}, e_2^{\{v_5,v_6\}}$
v_3
v_4
v_5
v_6

Figure 4.1: Totalizers for cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$ (left) and $\kappa_2 = \{v_3, v_4, v_5, v_6\}$ (right).

Now, consider what happens when the two overlapping cores of Example 4.1 are relaxed in OLL with totalizers. Figure 4.1 illustrates one example of how totalizers could be constructed for the cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$ and $\kappa_2 = \{v_3, v_4, v_5, v_6\}$. Notice that both totalizers in Figure 4.1 have a subtree with leaves v_3 and v_4 (coloured red in Figure 4.1). These subtrees can be seen as equivalent in the sense that they contain variables with equivalent semantics: $\tau(d_0^{\{v_3,v_4\}}) = \tau(e_0^{\{v_3,v_4\}})$ and $\tau(d_1^{\{v_3,v_4\}}) = \tau(e_1^{\{v_3,v_4\}})$ holds for every satisfying assignment τ. This being the case, constructing these two totalizers separately would introduce duplicate clauses and variables having exactly the same semantics. The structure-sharing technique aims to avoid introducing this type of redundancy in cardinality constraints constructed over overlapping cores. In particular, since the cardinality constraint encodings we consider can be viewed as binary trees, structure-sharing aims at sharing subtrees between the tree structures introduced by cardinality constraints arising from extracted cores. Given any two trees that have a subtree with common input variables, the subtree can be shared between the trees. The correctness of structure-sharing follows straightforwardly from the fact that it does not alter the semantics of the output variables.

Remark. *A form of structure-sharing has been proposed in a different context in [27]. In particular, the authors of [27] propose a form of structure-sharing in the context of answer set programming (ASP) aiming at compacting the CNF representation of the cost function. Whereas in [27] structure-sharing is employed to statically before search to encode* a single pseudo-Boolean constraint*, we apply structure-sharing to a set of cardinality constraints constructed over a set of cores iteratively extracted during core-guided search.*

4.1.1 Structure-sharing in OLL and PMRES

We now detail how to apply structure-sharing to totalizers of OLL and cardinality constraints of PMRES.

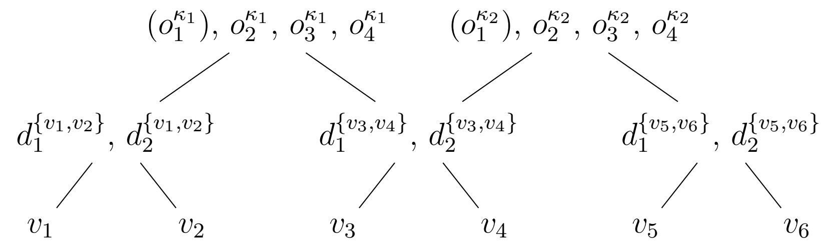

$(o_1^{\kappa_1})$, $o_2^{\kappa_1}$, $o_3^{\kappa_1}$, $o_4^{\kappa_1}$
$(o_1^{\kappa_2})$, $o_2^{\kappa_2}$, $o_3^{\kappa_2}$, $o_4^{\kappa_2}$
$d_1^{\{v_1,v_2\}}$, $d_2^{\{v_1,v_2\}}$
$d_1^{\{v_3,v_4\}}$, $d_2^{\{v_3,v_4\}}$
$d_1^{\{v_5,v_6\}}$, $d_2^{\{v_5,v_6\}}$
v_1
v_2
v_3
v_4
v_5
v_6

Figure 4.2: The structure of totalizers when relaxing the cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$ and $\kappa_2 = \{v_3, v_4, v_5, v_6\}$ with structure-sharing.

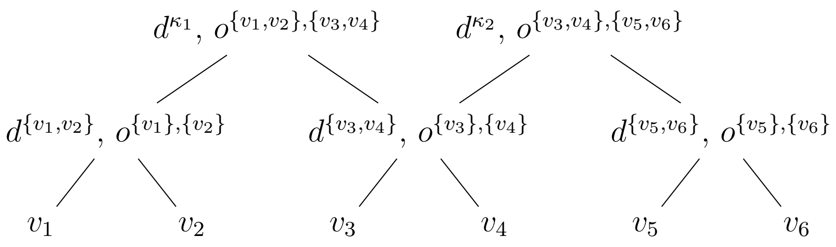

d^{κ_1}, $o^{\{v_1,v_2\},\{v_3,v_4\}}$
d^{κ_2}, $o^{\{v_3,v_4\},\{v_5,v_6\}}$
$d^{\{v_1,v_2\}}$, $o^{\{v_1\},\{v_2\}}$
$d^{\{v_3,v_4\}}$, $o^{\{v_3\},\{v_4\}}$
$d^{\{v_5,v_6\}}$, $o^{\{v_5\},\{v_6\}}$
v_1
v_2
v_3
v_4
v_5
v_6

Figure 4.3: The structure of PMRES cardinality constraint when relaxing cores $\{v_1, v_2, v_3, v_4\}$ and $\{v_3, v_4, v_5, v_6\}$ with structure-sharing.

Including structure-sharing in an implementation of OLL using totalizers is relatively straightforward whereas the inclusion in PMRES requires more care. An example of how the totalizers of OLL can share a subtree is illustrated in Figure 4.2. In particular, Figure 4.2 illustrates how totalizers for cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$ and $\kappa_2 = \{v_3, v_4, v_5, v_6\}$ of the Example 4.1 share a subtree for their common variables v_3 and v_4. In the resulting structure, there are two root nodes, one for each core. Both root nodes have two child nodes. The root for κ_1 has a child node for input variables v_1 and v_2, and another child node for input variables v_3 and v_4. The child node for input variables v_3 and v_4 is also linked as a child of the root node for κ_2. The other child of the root node for κ_2 has as its input variables v_5 and v_6.

Figure 4.3 illustrates an example of PMRES cardinality constraints with structure-sharing for the cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$ and $\kappa_2 = \{v_3, v_4, v_5, v_6\}$ of Example 4.1. The main idea is the same as for totalizers in OLL. A shared subset of variables can be inserted into a subtree that is encoded only once and then linked to both structures. However, when incorporating structure-sharing into PMRES, a few additional issues need to be dealt with.

First, since in the encoding used by PMRES, each inner node has an output variable, there will be output variables in the shared subtree. Because these output variables are outputs for more than one core, the initialization of their weights must be adjusted accordingly.

Specifically, the weight of an output variable is initialized to the sum of the minimum weights of the cores for which the output variable is an output.

Second, efficient structure-sharing between multiple overlapping cores requires generalized PMRES (recall Section 3.3). The reason for this is that in the original chain-like structure the sets of input variables of subtrees rooted on the inner nodes are never disjoint. Instead, for any two subtrees (excluding leaves), the set of input variables of one is always a subset of the set of the input variables of the other. To understand why this is a problem, consider the following example.

Example 4.2

Assume that an execution of PMRES extended with structure-sharing is at a stage where it is about to construct cardinality constraints for the cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$, $\kappa_2 = \{v_1, v_2, v_5, v_6\}$, $\kappa_3 = \{v_3, v_4, v_7, v_8\}$ obtained by WCE. Potentially, κ_1 could share a subtree for $\{v_1, v_2\}$ with κ_2 and a subtree for $\{v_3, v_4\}$ with κ_3. However, the chain-like structure does not allow constructing separate subtrees for both $\{v_1, v_2\}$ and $\{v_3, v_4\}$ in a cardinality constraint for κ_1. Generalized PMRES resolves this problem: when the structure of the cardinality constraint can be any binary tree, separate subtrees can be constructed for any disjoint sets of input variables.

Even though structure-sharing is a conceptually simple idea, realizing the technique is not as straightforward. In the following, we first discuss why WCE is a key for enabling structure-sharing and then address the problem of how to select which subsets of variables in cores to share.

4.1.2 The role of WCE

Without WCE, the structure of a cardinality constraint is selected without any knowledge about which variables will be common with future cores. The selected structure might not enable structure-sharing.

Example 4.3

Consider the cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$ and $\kappa_2 = \{v_3, v_4, v_5, v_6\}$ from Example 4.1. Without WCE, a core-guided algorithm could first find a core κ_1 and then construct a cardinality constraint over κ_1 without any knowledge of which variables will be in the κ_2. In Figure 4.4 we have illustrated another possible totalizer for κ_1 from Example 4.1. The structure of the totalizer in Figure 4.4 does not enable structure-

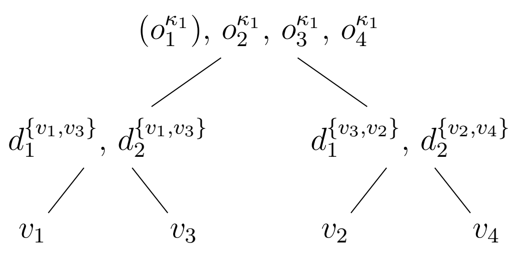

$(o_1^{\kappa_1}), o_2^{\kappa_1}, o_3^{\kappa_1}, o_4^{\kappa_1}$
$d_1^{\{v_1,v_3\}}, d_2^{\{v_1,v_3\}}$
$d_1^{\{v_3,v_2\}}, d_2^{\{v_2,v_4\}}$
v_1
v_3
v_2
v_4

Figure 4.4: A possible totalizer for a core $\kappa_1 = \{v_1, v_2, v_3, v_4\}$ that does not enable structure-sharing with $\kappa_2 = \{v_3, v_4, v_5, v_6\}$.

sharing with $\kappa_2 = \{v_3, v_4, v_5, v_6\}$, since the common variables v_3 and v_4 are not in the same subtree.

It is relatively easy to see that it is actually highly unlikely that a randomly selected structure for a cardinality constraint would enable efficient structure-sharing: when constructing a cardinality constraint for a core κ, only a small number of the $|\kappa|!$ possible permutations of how to order the input variables allows for setting the desired input variables to the same subtree. Furthermore, even when the order of the input variables would allow this, only in a small number of the possible trees realizing that order of input variables the input variables will be in the same subtree.

WCE alleviates this problem: the structure of the cardinality constraints can be selected with a knowledge of which variables are common between different cores in the set that is being relaxed.

4.1.3 Selecting what to share

Even with WCE, an efficient realization of structure-sharing—in particular, selecting which subsets of variables to share with which constraints—is not straightforward. The following example illustrates this selection problem.

Example 4.4

Consider the case with three cores $\kappa_1 = S_1 \cup \{x, y, z\}$, $\kappa_2 = S_2 \cup \{x, y\}$, $\kappa_3 = S_3 \cup \{y, z\}$, where the S_i are any sets of distinct variables. Core κ_1 has variables $\{x, y\}$ common with κ_2 and variables $\{y, z\}$ common with κ_3. In this case, we could potentially construct a subtree for $\{x, y\}$, to be shared between κ_1 and κ_2. However, then we can not construct a subtree for $\{y, z\}$ in the totalizer for κ_1 to be shared with κ_3.

Example 4.4 illustrates that it is not always possible to realize shared structures for all variables shared by different cores. More generally, when constructing cardinality constraints with structure-sharing, one needs to make choices on which shared subtrees to realize from all potential ones. Ideally, this would be done in a way that enables as much sharing as possible. However, selecting an optimal structure is not straightforward. In fact, it is not even clear what should be optimized: the number of shared variables or the number of shared clauses. A crucial issue is that the number of possible ways of selecting which subsets to share appears to be—at the worst case—exponential to the number of cores as illustrated by the following example.

Example 4.5

Consider the case where there are n cores, $\kappa_1 = \{v_1, v_2, \ldots, v_{n+1}\}, \kappa_2 = \{v_2, v_3, \ldots, v_{n+2}\}, \kappa_3 = \{v_3, v_4, \ldots, v_{n+3}\}, \ldots, \kappa_n = \{v_n, v_{n+1}, \ldots v_{n+n}\}$. A subtree for $\{v_n, v_{n+1}\}$ could be shared with all totalizers. Then, one could construct a subtree either for $\{v_n, v_{n+1}, v_{n+2}\}$ or $\{v_{n-1}, v_n, v_{n+1}\}$ and share it with $n-1$ totalizers. From both cases, one could again extend the shared subtree by one variable (with 2 choices of which variable to include) and share it with $n-2$ totalizers. This leads to 2^n ways of selecting which subsets to share, and this does not even include all possible ways of selecting the subsets.

Also notice that—in the case of totalizers—neither the number of shared variables nor the number of shared clauses is linear to the number of shared inputs, which means that to minimize the combined size of the totalizers it is not sufficient to maximize the number of shared input variables between the totalizers. And finally, we point out that the way in which the shared subsets of variables are selected also has an impact on the structures of totalizers. If one wishes to minimize the total number of variables in the structure, one should not only consider how many variables can be shared but also how balanced the totalizers are. (Recall that the number of variables in a totalizer depends on how balanced the totalizer is.)

All in all, we conjecture that selecting optimally what to share is NP-hard. Instead of aiming for optimal sharing, we propose a greedy algorithm described next.

4.1.4 A greedy algorithm for selecting shared subtrees

Algorithm 4 provides a description of our greedy approach for selecting which subtrees to share between which cardinality constraints in pseudocode. The approach is applicable

Algorithm 4: Build-Card-Constraints-Greedy an algorithm for selecting which subsets of variables to share between which cores.

```
Input: A set of cores $\mathcal{K}$
Output: A set of totalizers for $\mathcal{K}$
1  Nodes ← Init-Nodes($\mathcal{K}$)
2  while 1 do
3      $(N_1, N_2) \leftarrow \arg\max_{N_i, N_j \in \text{Nodes}} |N_i.\text{vars} \cap N_j.\text{vars}|$
4      $C \leftarrow N_1.\text{vars} \cap N_2.\text{vars}$
5      if ($|C| <$ THRE) then break
6      $N \leftarrow$ Init-Node($C$)
7      $N_1.\text{vars} \leftarrow N_1.\text{vars} \setminus C$
8      $N_2.\text{vars} \leftarrow N_2.\text{vars} \setminus C$
9      $N_1$.addDescendant($N$)
10     $N_2$.addDescendant($N$)
11     Nodes.add($N$)
12 return Build-Constraints(Nodes)
```

both in OLL and PMRES. The main idea of the greedy algorithm is to iteratively select one of the largest possible shared subtrees to be realized. An intuition behind the idea is that sharing as large substructures as possible maximizes relatively well the total size of shared structures (recall that in totalizers, the size of a subtree is non-linear to the number of input variables). Given a set $\mathcal{K} = \{\kappa_1, \ldots, \kappa_n\}$ of cores for which to construct the constraints, the algorithm decides which subsets of variables in these cores to share between which cardinality constraints. The algorithm produces a skeleton for the structure of the cardinality constraints by producing a set of nodes Nodes to be realized in the cardinality constraints. Each node N of Nodes is associated with a set of variables that will be in the leaves of the subtree rooted on N in the final structure. Each node N is also associated with a set of nodes that will be descendants of N in the final structure.

The algorithm begins by initializing Nodes on Line 1. The initial nodes in Nodes correspond to the roots of the totalizers in the final structure and each of them is associated with the set of variables in the corresponding core. In the main loop (Lines 2–11) the following process is iterated. A pair of Nodes that has a maximum number of common variables is selected (Line 3). A new node for the common variables is introduced (Line 6). The new node corresponds to a root of a subtree that is to be shared. The common variables are removed from the pair (Lines 7 and 8), and the new node is linked as a descendant to both of them with addDescendant function (Lines 9 and 10). Finally, the new node is also added to the set of nodes (Line 11).

The main loop terminates when the maximum number of common variables between any two nodes is below THRE, a user-defined constant (Line 5). The reason for using a threshold value to terminate the search is to avoid using too much time in selecting which subsets to share. The following example provides an example execution of Algorithm 4.

Example 4.6

Consider the case where the algorithm Build-Card-Constraints-Greedy using THRE $= 2$ is executed for three cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$, $\kappa_2 = \{v_2, v_3, v_4, v_5, v_6\}$, $\kappa_3 = \{v_3, v_4, v_5, v_6, v_7\}$. The first step is to initialize the set Nodes. Denoting each node as a pair—the set of associated variables and the set of nodes linked as descendants—Nodes is initialized as

$$\begin{aligned}\textsc{Nodes} = \{N_1 = \langle\{v_1, v_2, v_3, v_4\}, \{\}\rangle, \qquad & N_2 = \langle\{v_2, v_3, v_4, v_5, v_6\}, \{\}\rangle,\\ N_3 = \langle\{v_3, v_4, v_5, v_6, v_7\}, \{\}\rangle\}.\end{aligned}$$

The execution of the main loop begins with finding a pair of nodes that have a maximum number of common associated variables. The nodes N_2 and N_3 have four common associated variables, v_3, v_4, v_5 and v_6. This is the maximum number among any pair of nodes. After introducing a new node N_4 and updating the sets of associated variables and linked descendants, the new set of nodes is

$$\begin{aligned}\textsc{Nodes} = \{N_1 = \langle\{v_1, v_2, v_3, v_4\}, \{\}\rangle, \qquad & N_2 = \langle\{v_2\}, \{N_4\}\rangle,\\ N_3 = \langle\{v_7\}, \{N_4\}\rangle, \qquad & N_4 = \langle\{v_3, v_4, v_5, v_6\}, \{\}\rangle\}.\end{aligned}$$

Now the nodes N_1 and N_4 have a maximum number of common associated variables (v_3 and v_4) among any pair. After introducing a new node N_5 and updating the sets of associated variables and linked descendants, the set of nodes is

$$\begin{aligned}\textsc{Nodes} = \{N_1 = \langle\{v_1, v_2\}, \{N_5\}\rangle, \qquad & N_2 = \langle\{v_2\}, \{N_4\}\rangle,\\ N_3 = \langle\{v_7\}, \{N_4\}\rangle, \qquad & N_4 = \langle\{v_5, v_6\}, \{N_5\}\rangle,\\ N_5 = \langle\{v_3, v_4\}, \{\}\rangle\}. \qquad &\end{aligned}$$

Since now there are no more pairs of nodes that have more than one variable in common, the search terminates. The final step of Build-Card-Constraints-Greedy

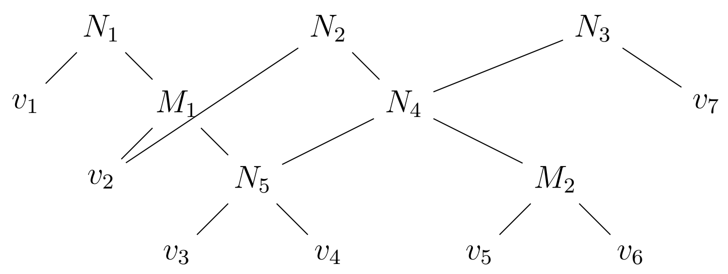


Figure 4.5: An example of more complicated structure-sharing from Example 4.6 for cores $\kappa_1 = \{v_1, v_2, v_3, v_4\}$, $\kappa_2 = \{v_2, v_3, v_4, v_5, v_6\}$, $\kappa_3 = \{v_3, v_4, v_5, v_6, v_7\}$.

is to use the set NODES to construct the cardinality constraints. An example of the final structure of cardinality constraints corresponding to the selected shared subtrees of this example is illustrated in Figure 4.5.

4.2 Selective introduction equivalence-defining clauses

We will now detail how structure-sharing can enable additional unit propagations in non-trivial ways. We begin by overviewing how cardinality constraints are encoded into CNF and how partial assignments unit propagate values on the constraints. Then we will propose our own approach of using a heuristic algorithm to encode the cardinality constraints in a way that balances between the additional propagation properties and the size of the encoding. In the following, we focus on OLL and totalizers. Similar observations could also be made for the cardinality constraints of PMRES.

4.2.1 Equivalences and implications on totalizer encodings

The encoding of the semantics of the variables of a totalizer is defined in a recursive manner [20]. We label the nodes of a totalizer by the set of inputs in the subtree. Let $L \cup R$ be an internal node with inputs $L \cup R$: L being a set of the inputs of the left child and R being the set of the inputs of the right child. Recall that the intended semantics of each variable in a totalizer is $d_i^S \leftrightarrow (\sum_{v \in S} v \geq i)$. The following clauses between the variables are included in the totalizer encoding for each $1 \leq i \leq |L|$ and $1 \leq j \leq |R|$.

1. $(\neg d_i^L \vee d_i^{L \cup R})$, encoding $d_i^L \rightarrow d_i^{L \cup R}$

2. $(\neg d_j^R \vee d_j^{L \cup R})$, encoding $d_j^R \to d_j^{L \cup R}$
3. $(\neg d_i^L \vee \neg d_j^R \vee d_{i+j}^{L \cup R})$, encoding $(d_i^L \wedge d_j^R) \to d_{i+j}^{L \cup R}$
4. $(d_i^L \vee \neg d_{i+|R|}^{L \cup R})$, encoding $\neg d_i^L \to \neg d_{i+|R|}^{L \cup R}$
5. $(d_j^R \vee \neg d_{j+|L|}^{L \cup R})$, encoding $\neg d_j^R \to \neg d_{j+|L|}^{L \cup R}$
6. $(d_i^L \vee d_j^R \vee \neg d_{i+j}^{L \cup R})$, encoding $(\neg d_i^L \wedge \neg d_j^R) \to \neg d_{i+j}^{L \cup R}$

For understanding unit propagation in the totalizers, consider the following observations.

First, the clauses included in the CNF encoding of totalizers are either binary clauses or ternary clauses. Each binary clause connects a node to one of its children. The ternary clauses connect a node to both of its children. We will describe how binary and ternary clauses unit propagate values in totalizers in Section 4.2.2.

Second, let us divide the clauses in the CNF encoding of totalizers into two groups based on the direction of the implication. The first group consists of clauses 1–3 (implications from positive variables of children to positive variables of the parent) and the second group consists of clauses 4–6 (implications from negated variables of children to negated variables of the parent).

When we described the semantics of the output variables of the cardinality constraints of OLL (recall Section 3.2), we defined the output variables as being equivalent to defined constraints over the input variables, i.e., $o_i^\kappa \leftrightarrow (\sum_{v \in \kappa} v \geq i)$. We refer to this as the *equivalence semantics*. However, for the correctness of the algorithm it is sufficient to enforce implication relationship, i.e., $(\sum_{v \in \kappa} v \geq i) \to o_i^\kappa$. This we refer to as the *implication semantics*. An intuition for why implication semantics is sufficient for correctness follows from the way the cardinality constraints are used: They are only used to enforce upper bounds for the number of variables assigned to 1 in the core.

Encoding a totalizer with implication semantics introduces clauses of type 1–3 for each node. We call these clauses the *implication-defining clauses*. To encode a totalizer with the equivalence semantics, one also needs to introduce clauses 4–6, which is why we call these additional clauses *equivalence-defining clauses*. When encoding a totalizer in the context of OLL, implication-defining clauses are added to every node, whereas equivalence-defining clauses may be included optionally at will.

Encoding the cardinality constraints with implication semantics rather than with equivalence semantics reduces the number of needed clauses by half. Most implementations of

OLL that we are aware of (including [45, 71, 14]) add only implication-defining clauses as this keeps the size of the encoding smaller. However, as we will see next, the equivalence-defining clauses enable the internal SAT solver to do more reasoning which, as we will argue, may be useful in some cases especially when structure-sharing is used.

4.2.2 Unit propagation in totalizers

We now turn to the question of how values are unit propagated within a SAT solver in the totalizers, i.e., how, given a partial assignment of values to the variables in the totalizers, values are unit propagated to other yet unassigned variables in the structure. The implication-defining clauses propagate 1s from nodes to their parents and 0s from parents to their children. The equivalence-defining clauses propagate 0s from nodes to their parents and 1s from parents to their children. In addition to this distinction, the propagation in totalizers can be classified according to whether it is caused by binary or ternary clauses. Binary clauses cause propagation where one value propagates a value to another variable. Ternary clauses cause propagation where two given values propagate a value to the third variable of the clause.

The different cases of child-to-parent propagation are illustrated in Figure 4.6. The single arrows represent propagation by binary clauses, and two lines combining into one arrow represent propagation by ternary clauses. For example, the left-most blue arrow in Figure 4.6 implies that when the left-most child is assigned to 1, the variable on its parent is also propagated to 1. The blue two-tailed arrow in the figure implies that when the variables at the tails are assigned to 1s, the variable at the head of the arrow in the root is also propagated to 1. The question marks in the illustrations represent variables that are not yet assigned values. Simple cases of parent-to-child propagation, where binary clauses propagate values are illustrated in Figure 4.7. Cases of parent-to-child propagation where also ternary clauses propagate values are illustrated in Figure 4.8.

4.2.3 Equivalences and structure-sharing

We will now detail how equivalences combined with structure sharing enables interesting unit propagations in totalizers.

In the context of OLL, propagation in totalizers can be considered of interest if a value is propagated to either an output or an input variable. The reason for this is that only the

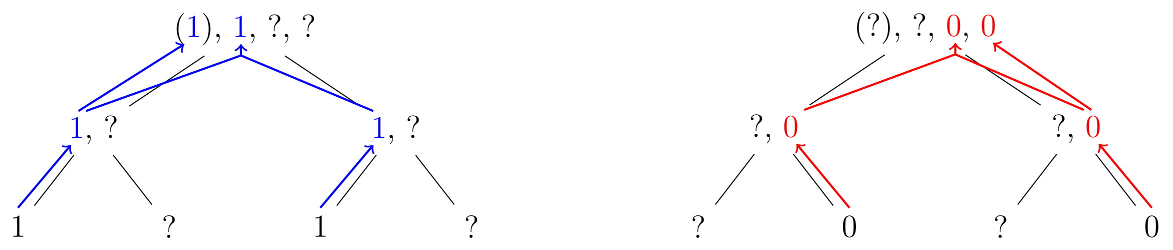


Figure 4.6: Child-to-parent propagation on totalizers via implication-defining clauses (left) and equivalence-defining clauses (right). The values that are fixed by the partial assignment are black, the values propagated by implication-defining clauses are blue, and the values propagated by the equivalence-defining clauses are red.

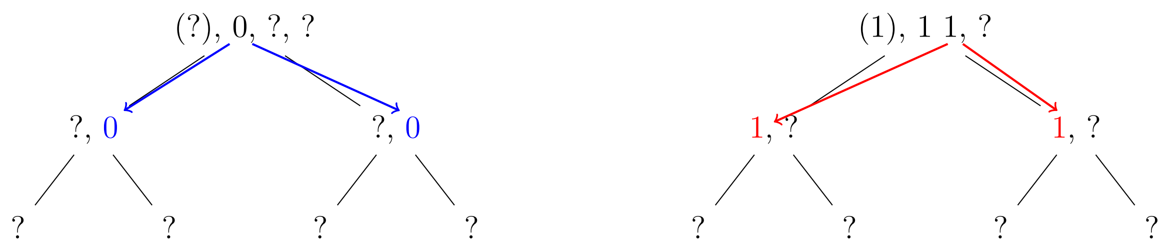


Figure 4.7: Parent-to-child propagation by binary implication-defining clauses (left) and binary equivalence-defining clauses (right).

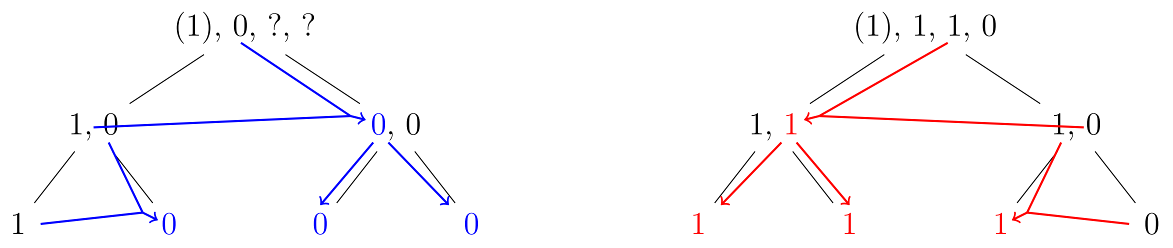


Figure 4.8: Parent-to-child propagation by both, binary and ternary clauses. Propagation by implication-defining clauses (left) and equivalence-defining clauses (right).

values of input and output variables are meaningful for the actual MaxSAT instance being solved. The inner variables are only auxiliary variables to encode the relationship between the input and output variables: their values do not have any meaningful implications outside the totalizer, and unlike input and output variables, they are never mentioned in any clause outside the totalizer.

Structure-sharing somewhat changes the situation. When a number of totalizers share internal variables, propagating values on these shared internal variables may cause unit propagation to the outputs of other totalizers sharing the same subtree. An example of this is shown in Figure 4.9. The values on the left-hand-size totalizer propagate a value

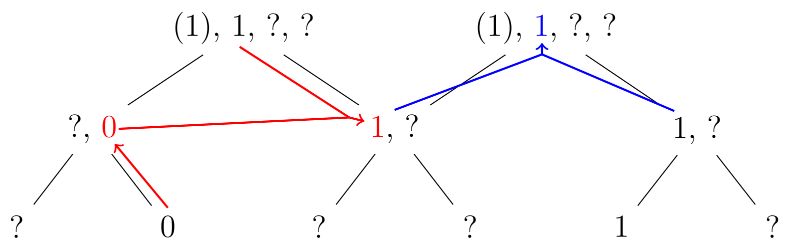


Figure 4.9: Reasoning between cardinality constraints with equivalence-defining clauses and structure-sharing.

on the shared subtree of the totalizers, and this value then contributes to propagating a value on an output variable of the right-hand-side totalizer to propagate. Notice that this unit propagation requires structure-sharing and both implication and equivalence-defining clauses.

An intuition of why implication and equivalence-defining clauses are both needed follows from the following observation. To propagate a value via a shared subtree that could not be propagated without structure-sharing, the propagated value must originally come from a parent of the shared node (otherwise the value would be "known" in all totalizers sharing the node even without structure-sharing). The value must then be propagated from the shared node to (one of the) other parents. A value being propagated first in parent-to-child direction and then child-to-parent direction requires both implication and equivalence-defining clauses. The reason for this is that implication-defining clauses propagate 0s in the parent-to-child direction only, and 1s in the child-to-parent direction only, while equivalence-defining clauses propagate similarly but in the opposite direction.

4.2.4 A heuristic for introducing equivalence-defining clauses

Based on the observations that equivalence-defining clauses increase the size of the cardinality constraint encoding but on the other hand may induce additional propagations, we propose a heuristic algorithm for adding equivalence-defining clauses sparingly to specific parts of the cardinality constraints. The aim is to balance between the number of clauses introduced and the propagations enabled. Our algorithm estimates how likely adding equivalence-defining clauses to some parts is useful and how much it increases the size of the encoding. We consider an equivalence-defining clause useful when it contributes propagating value 1 to either an input variable or to a variable on a shared subtree. For each node in which equivalence-defining clauses could induce this type of propagation we estimate two parameters:

1. *Cost:* How many equivalence-defining clauses should be added in total to enable the propagation.

2. *Likelihood of being useful*: How many input variables must be assigned to 1 before the useful propagation can take place.

The algorithm introduces equivalence-defining clauses when both values (1) and (2) are below a user-defined parameter.

5 Empirical evaluation

In this chapter, we present an overview of the result from an empirical evaluation of the proposed structure-sharing technique. We evaluated the impact of structure-sharing on two different implementations of OLL and PMRES. The results suggest that incorporating structure-sharing improves solver runtimes and reduces the number of clauses and variables in the cardinality constraint encodings in practice. We also provide empirical data on the impact of the equivalence clauses on solver runtimes.

5.1 Implementation

We incorporated the structure-sharing technique into two implementations of OLL and two implementations of PMRES.[2] First, we incorporated structure-sharing into a modified version of a state-of-the-art implementation of OLL using totalizers, RC2 [45], which is built on PySAT [44]. We call this solver CGSS. We also extended the RC2-based implementation with PMRES. Furthermore, we reimplemented OLL and PMRES in C++ from scratch and incorporated structure-sharing into the new implementation. We call this solver CGSS2. The source codes of all implementations are available online. RC2 [45] was cloned from `https://github.com/pysathq/pysat`.[3] RC2-based CGSS [46] (our modification of RC2 geared towards enabling structure sharing and WCE) is available from `https://bitbucket.org/coreo-group/cgss`.[4] The novel C++ implementation, CGSS2, is available from `https://bitbucket.org/coreo-group/cgss2`.[5]

Our implementations use standard techniques in core-guided MaxSAT solving: the so-called stratification technique [5, 6, 56] hardening [6, 67], so-called core trimming [68, 45], core minimization [45, 60] and core exhaustion techniques [45, 6]. Totalizers are constructed incrementally [64, 63, 62]. As the underlying SAT solver, we used Glucose 3.0 [13] in the empirical evaluation. The PySAT-based implementations support various SAT solvers and CGSS2 supports CaDiCal [25] in addition to Glucose 3.0. However, based on preliminary tests, Glucose 3.0 seemed to result in better runtime performance overall.

[2] An empirical evaluation of the RC2-based CGSS was published in [46].

[3] The version of commit b7ac61b0830ab989159a3cd37269fe97916eb325 was used.

[4] The version of commit 0ca4dcc57b88f834477a349cfd1ffd3c5ca2457c was used.

[5] The version of commit 28ca2eac99de1e03056a5b771c50d3cfdb2f4d96 was used.

We ran preliminary tests to select a threshold value for terminating the loop of our implementation of the algorithm Build-Card-Constraints-Greedy. We decided to use THRE = 16 since it resulted in slightly better runtime performance than other values tested. However, it is possible that further tuning THRE could result in further improvements. For adding equivalence constraints we used 50 as a threshold value for both of the estimated values, *Cost* and *Likelihood of being useful* (recall Section 4.2.4). In preliminary tests, the impact of different values appeared not very noticeable.

All experiments reported in the following sections were run single-threaded using 2.6-GHz Intel Xeon E5-2670 processors. A per-instance time limit of 3600 seconds and a memory limit of 32 GB were enforced. As benchmarks, we used the 989 instances combined from the complete weighted track of 2019 and 2020 MaxSAT Evaluations [18, 15] with duplicate instances removed. The benchmarks were obtained online from `https://maxsat-evaluations.github.io/`.

5.2 Results

In this section, we present an overview of the results of our empirical evaluation.

Table 5.1 details the solver variants used in the tests. For reproducibility, we list the exact command line options needed to run each of the variants used in the experiments in Appendix Table B.1.

5.2.1 Impact of structure-sharing

To analyze the impact of structure-sharing on the runtime performance, we ran experiments without WCE or structure-sharing, with WCE only, and with WCE and structure-sharing (SS) both enabled. Figure 5.1 illustrates the impact of the techniques on our two implementations of OLL. It shows the number of solved instances (out of 989) as a function of a per-instance time limit. First, we point out that our modifications to RC2 improve its performance. **OLL/CGSS** solves 701 instances while **OLL/RC2** solves 696. Enabling WCE improves the performance further. **OLL+WCE/CGSS** solves 704 instances while **OLL/CGSS** solves 701. Finally, the variant with structure-sharing performs better than the variant without. **OLL+WCE+SS/CGSS** solves 711 instances while **OLL+WCE/CGSS** solves 704.

Our C++ implementation outperforms the PySAT-based implementations regardless of

Table 5.1: Solver variants tested in the experimental evaluation

Variant name	**Explanation**
OLL+WCE+SS/CGSS2	C++ implementation of OLL with WCE and SS, heuristic addition of equivalence-defining clauses
OLL+WCE+SS/CGSS2, all eqs	C++ implementation of OLL with WCE and SS, add equivalence-defining clauses to all nodes
OLL+WCE+SS/CGSS2, no eqs	C++ implementation of OLL with WCE and SS, add implication-defining clauses only
OLL+WCE/CGSS2	C++ implementation of OLL with WCE
OLL/CGSS2	C++ implementation of OLL
OLL+WCE+SS/CGSS	PySAT-based implementation of OLL with WCE and SS, heuristic addition of equivalence-defining clauses
OLL+WCE+SS/CGSS, all eqs	PySAT-based implementation of OLL with WCE and SS, all equivalence-defining clauses to all nodes
OLL+WCE+SS/CGSS, no eqs	PySAT-based implementation of OLL with WCE and SS, add implication-defining clauses only
OLL+WCE/CGSS	PySAT-based implementation of OLL with WCE
OLL/CGSS	PySAT-based implementation of OLL (modified version of RC2)
OLL/RC2	Original RC2, PySAT-based implementation of OLL
PMRES+WCE+SS/CGSS2	C++ implementation of PMRES with WCE and SS
PMRES+WCE/CGSS2	C++ implementation of PMRES with WCE
PMRES/CGSS2	C++ implementation of PMRES
PMRES+WCE+SS/CGSS	PySAT-based implementation of PMRES with WCE and SS
PMRES+WCE/CGSS	PySAT-based implementation of PMRES with WCE
PMRES/CGSS	PySAT-based implementation of PMRES

which settings are fixed: the variant **OLL/CGSS2** outperforms the variant **OLL/CGSS** (707 compared to 701 solved instances), the variant **OLL+WCE/CGSS2** outperforms the variant **OLL+WCE/CGSS** (716 compared to 704 solved instances) and the variant **OLL+WCE+SS/CGSS2** outperforms the variant **OLL+WCE+SS/CGSS** (720 compared to 711 solved instances). Comparing the C++ variants of OLL with each other shows a similar trend to the comparison of PySAT-based variants of OLL. The variant with WCE performs better than the variant without and the variant with structure-

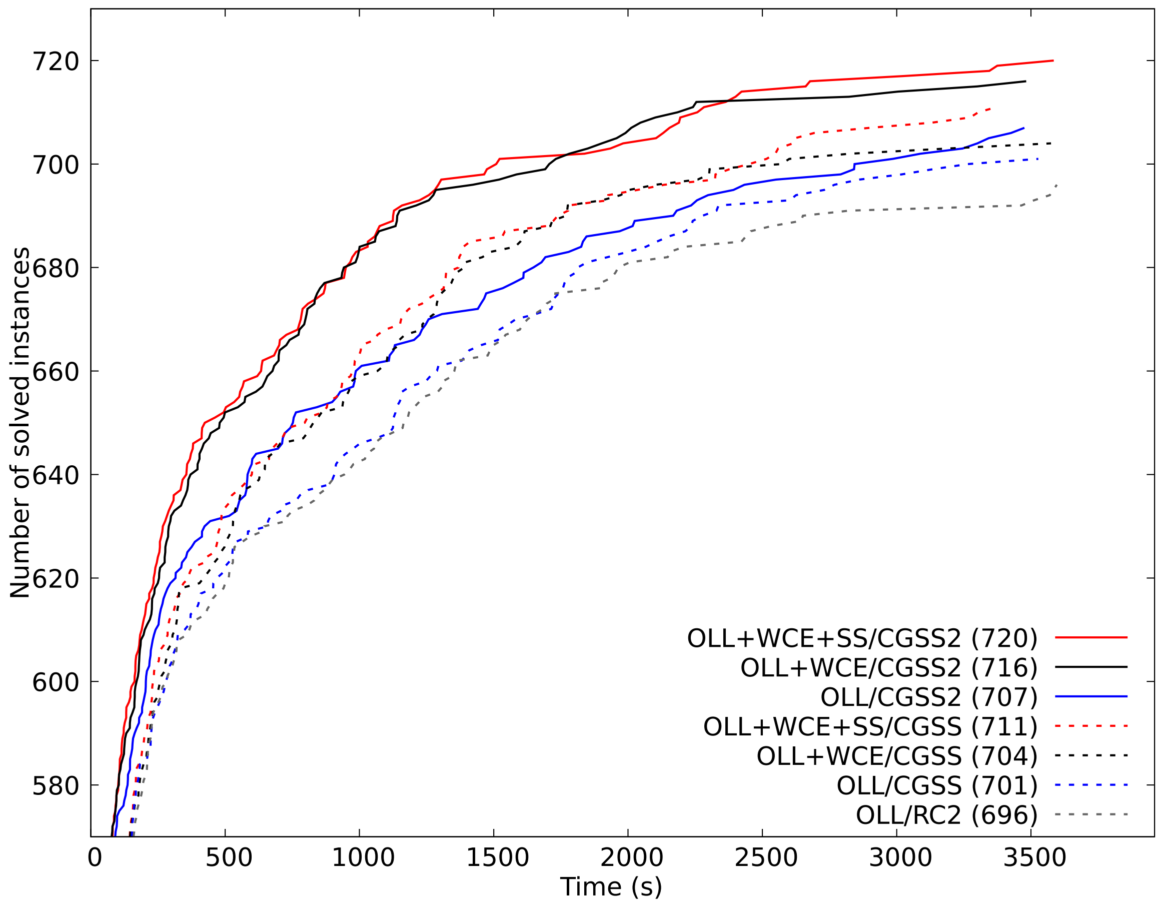


Figure 5.1: The impact of WCE and structure-sharing on our two implementations of OLL.

sharing performs better than the variants without. **OLL/CGSS2** solves 707 instances, **OLL+WCE/CGSS2** solves 716 instances and **OLL+WCE+SS/CGSS2** solves 720 instances. The impact of structure-sharing appears to be less pronounced in the C++ variant than in the PySAT-based variant.

We also ran similar experiments on our implementations of PMRES. The results are illustrated in Figure 5.2. A similar trend as in the results for OLL is visible here, both in PySAT-based and C++ implementations of PMRES.

The PySAT-based **PMRES+WCE/CGSS** solves more instances than its variant without WCE, **PMRES/CGSS** (629 instances compared to 622). Enabling structure-sharing results in still more solved instances: **PMRES+WCE+SS/CGSS** solves 644 instances while **PMRES+WCE/CGSS** solves 629. As in OLL, C++ variants perform better than PySAT-based variants regardless of which settings are fixed. **PMRES/CGSS2** solves 635 instances, **PMRES+WCE/CGSS2** 649 and **PMRES+WCE+SS/CGSS2** 653. The impact of structure-sharing appears to be more pronounced in PMRES than in OLL,

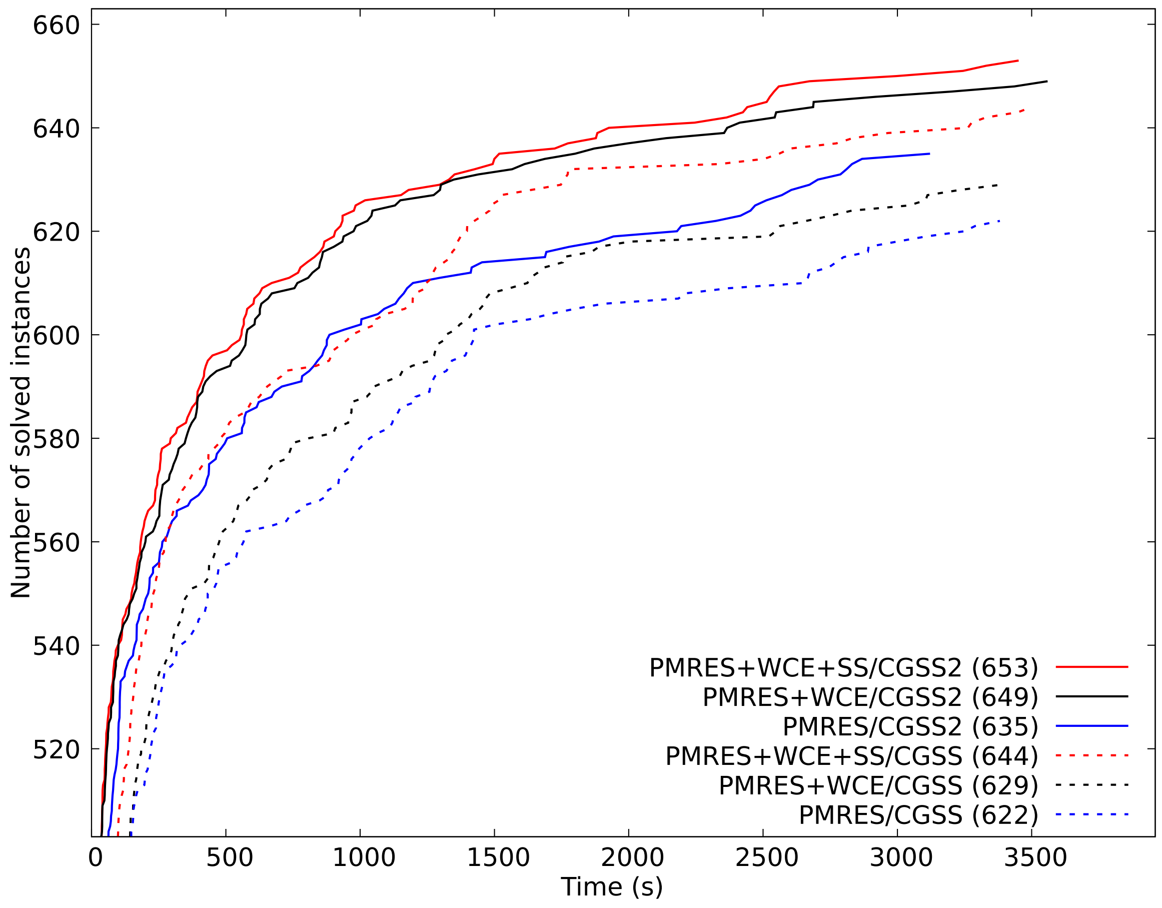


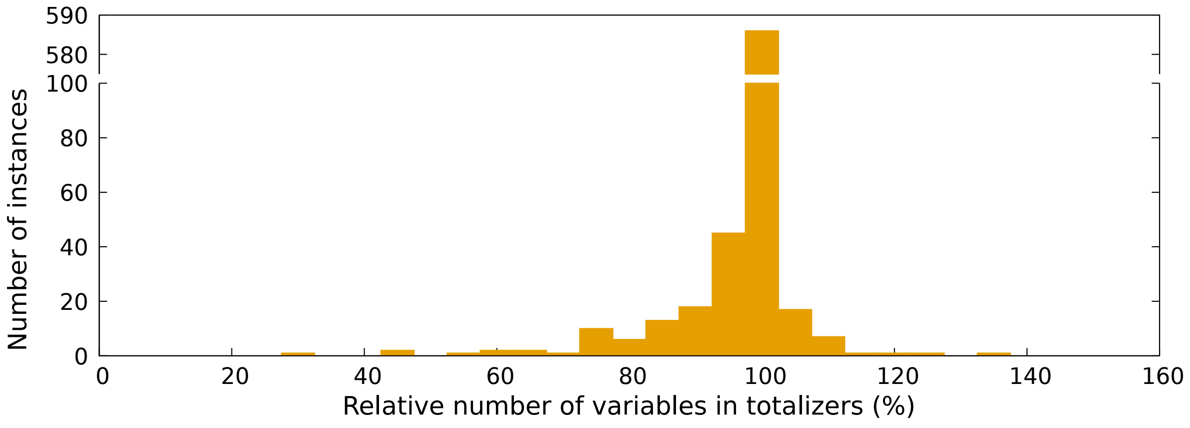
Figure 5.2: The impact of WCE and structure-sharing (SS) on our two implementations of PMRES.

especially in the PySAT-based variant.

The results demonstrate that our structure-sharing technique can improve the performance of core-guided solvers with soft cardinality constraints in practice.

We also analyze the impact of the structure-sharing technique on the size of the cardinality constraint encodings introduced during search. Figure 5.3 shows the impact of structure-sharing on the total number of variables and clauses in the totalizers of the final SAT instance. It provides a comparison of the final number of variables and clauses between variants **OLL+WCE+SS/CGSS2, no eqs** and **OLL+WCE/CGSS2** on the 716 instances solved within the time limit by both variants. The height of the bar at x-axis value p gives the number of instances for which the number of clauses (top) or variables (bottom) introduced by **OLL+WCE+SS/CGSS2** was $p\%$ of the number of clauses or variables introduced by **OLL+WCE/CGSS2**.

On a large number of instances, structure-sharing does not cause significant changes to the



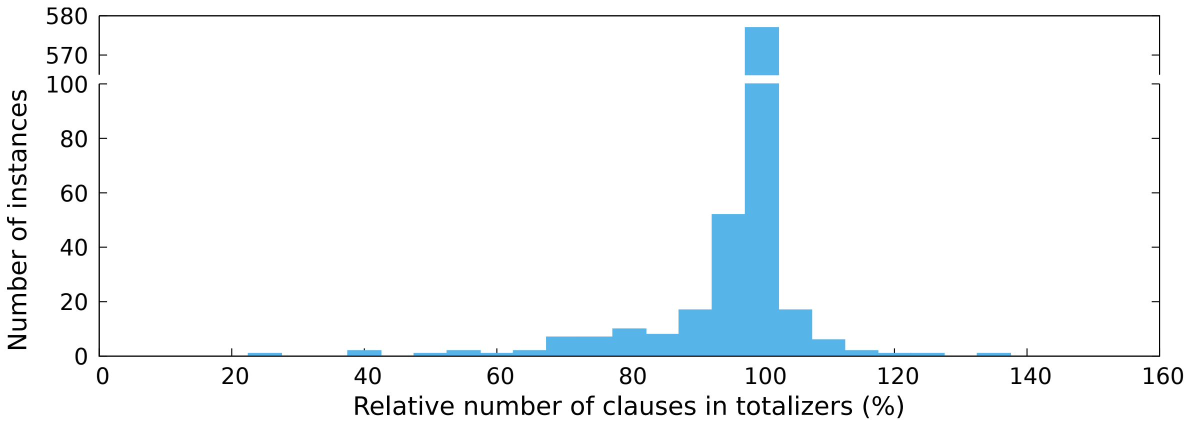
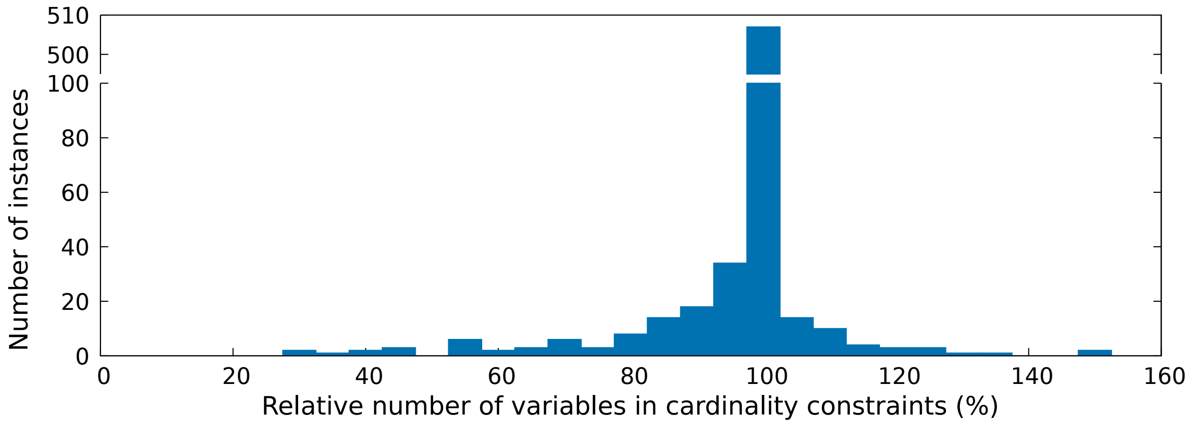


Figure 5.3: The impact of structure-sharing on the number of variables on totalizers (top) and the number of implication clauses (bottom), comparing **OLL+WCE+SS/CGSS2** and **OLL+WCE/CGSS2** on the 716 instances solved by both

number of clauses or variables. On 617 instances out of 716, the final number of clauses in totalizers of **OLL+WCE+SS/CGSS2** is 95%–105% of the final number of clauses in totalizers of **OLL+WCE/CGSS2**. When structure-sharing makes more difference, in most cases the result is fewer clauses and variables in totalizers. On 79 out of 716 instances, the variant with structure-sharing resulted in less than 95% clauses compared to the variant without, whereas on 20 instances resulted in more than 105% clauses. The most likely reason why sometimes the variant with structure-sharing technique ends up with larger formula is that the differences in the transformations to the formula have an impact on which cores the SAT solver returns. SAT solver may sometimes return larger cores which leads to larger cardinality constraint encodings. The number of variables follows a similar trend to the number of clauses.

Figure 5.4 shows similar data for the cardinality constraints of PMRES on the 648 instances



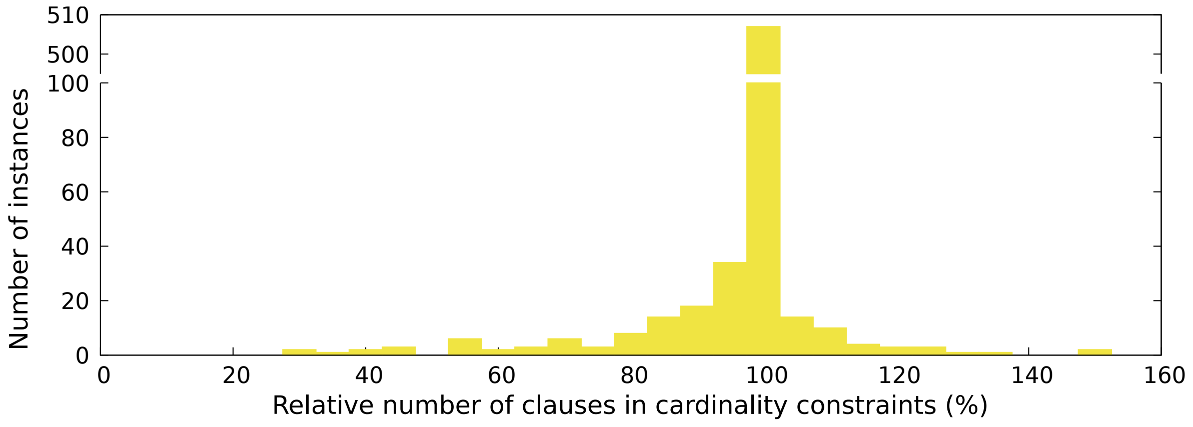


Figure 5.4: The impact of structure-sharing on the number of variables (top) and the number of clauses (bottom) of the cardinality constraints of PMRES, comparing **PMRES+WCE+SS/CGSS2** and **PM-RES+WCE/CGSS2** on the 648 instances solved by both.

solved by both **PMRES+WCE+SS/CGSS2** and **PMRES+WCE/CGSS2**. On 539 out of 648 instances the final number of clauses was 95%–105%. On 81 out of 648 instances the final number of clauses was less than 95% and on 28 out of 648 instances more than 105%. The number of variables follows a similar trend.

The results suggest that the structure-sharing technique has a notable impact on a relatively small number of instances. On a large number of instances, the impact of structure-sharing on the size of the cardinality constraints appears to be relatively small. When there is a larger impact, using structure-sharing more often results in a smaller encoding of cardinality constraints.

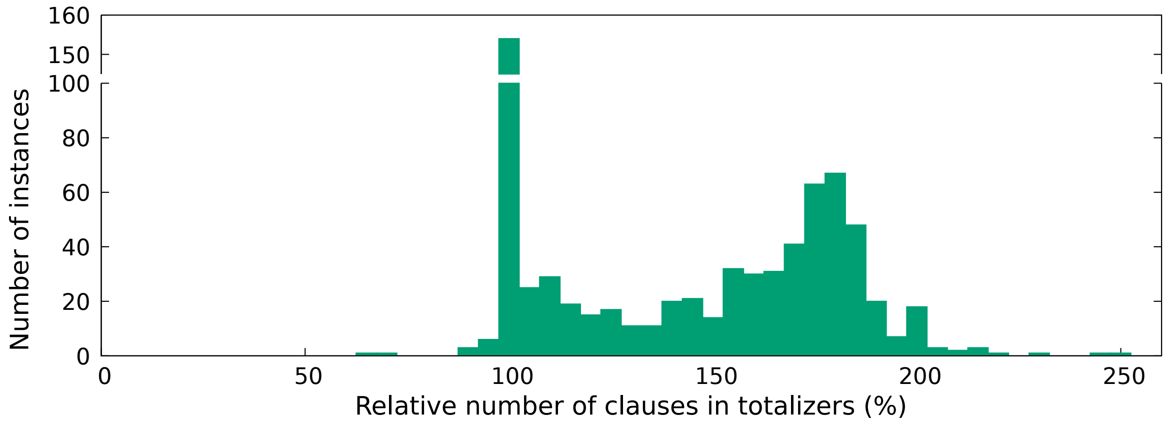


Figure 5.5: The impact of selective addition of equivalence constraints to the number of clauses compared to adding implication-defining clauses only.

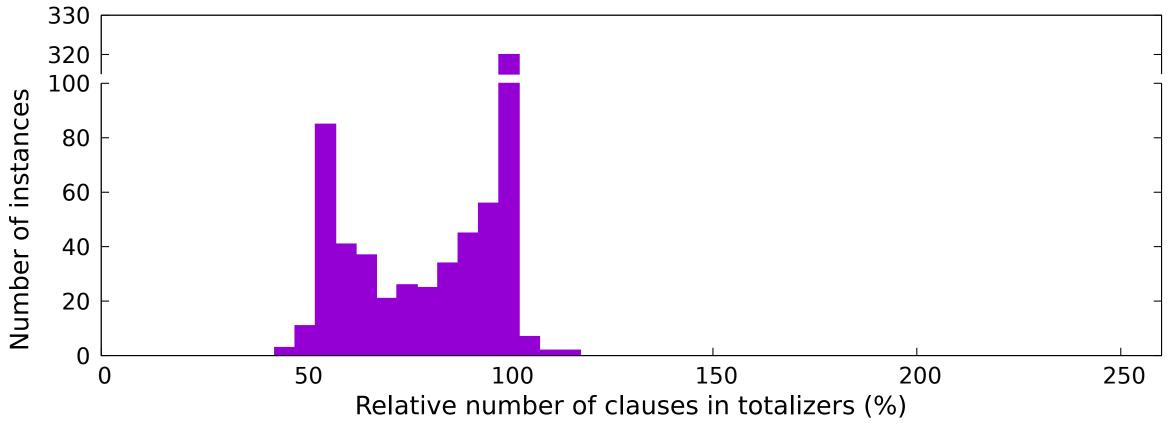


Figure 5.6: The impact of selective addition of equivalence-defining clauses to the number of clauses compared adding all equivalence-defining clauses.

5.2.2 Impact of the equivalence constraints

We also ran experiments to evaluate the impact of different strategies for adding equivalence-defining clauses on our implementations of OLL. We tested variants that added equivalence-defining clauses to every node, variants that added implication-defining clauses only and variants adding equivalence-defining clauses heuristically.

Figure 5.5 illustrates the impact of our heuristic for adding equivalence-defining clauses to the number of clauses in totalizers compared to a strategy of adding implication-defining clauses only. The height of the bar at x-axis value p gives the number of instances for which the number of clauses in cardinality constraints introduced by **OLL+WCE+SS/CGSS2** was $p\%$ of the number of clauses introduced by **OLL+WCE+SS/CGSS2, no eqs**. On a relatively large number of instances (170 out of 716) the number of clauses in the totalizers of **OLL+WCE+SS/CGSS2** is 95%–105% of the number in **OLL+WCE+SS/CGSS2, no eqs**. On other instances, the number of clauses in **OLL+WCE+SS/CGSS2** is mainly larger.

Figure 5.6 shows how the selective addition of equivalence-defining clauses impacts the final number of clauses compared to adding all equivalence-defining clauses. Again, the height of the bar at x-axis at value p gives the number of instances for which the number of clauses introduced by **OLL+WCE+SS/CGSS2** was $p\%$ of the number of implication-defining clauses introduced by **OLL+WCE+SS/CGSS2, all eqs**. On 348 out of 716 instances, the number of clauses introduced by **OLL+WCE+SS/CGSS2** is 95%–105% of the number clauses introduced by **OLL+WCE+SS/CGSS2, no eqs**. This suggests that in many cases the heuristic algorithm adds equivalence-defining clauses to every node. This could happen at least in cases where all the cores are small enough.

Another peak visible in Figure 5.6 is around 50%. The peak suggests that there is also a relatively large number of instances where the heuristic strategy adds a minimal number of equivalence-defining clauses. Recall that adding no equivalence-defining clauses at all results in an encoding that has half the clauses compared to adding all equivalence-defining clauses. Overall Figure 5.5 suggests that our strategy of adding equivalence-defining clauses does indeed find many situations where adding equivalence-defining clauses is considered useful. Figure 5.6 in turn, suggests that in many situations our strategy considers not adding equivalence-defining clauses a better option.

Figure 5.7 shows the performance of solver variants with implication-defining clauses only, all equivalence-defining clauses and the selective addition of equivalence-defining clauses. On PySAT-based variants, the selective addition technique appears to improve the performance. The variant that adds no equivalence-defining clauses (**OLL+WCE+SS/CGSS, no eqs**) solves 705 instances, the variant that adds equivalence-defining clauses to every node (**OLL+WCE+SS/CGSS, all eqs**) solves 709 instances and the variant that adds equivalence-defining clauses heuristically (**OLL+WCE+SS/CGSS**) solves 711 instances.

The results of PySAT-based variants suggest that our heuristic algorithm is successful. However, the results of C++ implementation somewhat challenge this. The variant with heuristic addition of equivalence-defining clauses (**OLL+WCE+SS/CGSS2**) solves only one instance more compared to a variant adding implication-defining clauses only

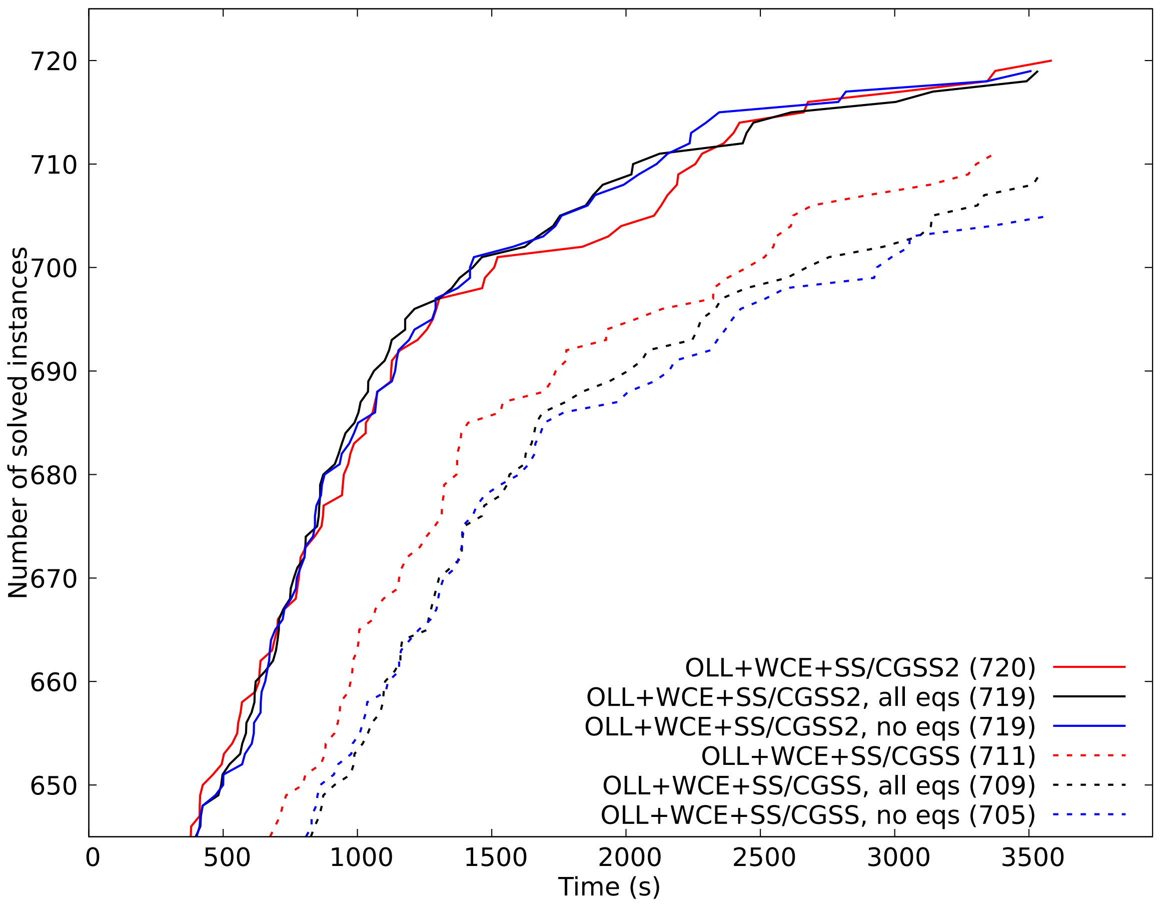


Figure 5.7: The impact of equivalence constraints on our two implementations of OLL.

(**OLL+WCE+SS/CGSS2, no eqs**) and to a variant adding all equivalence-defining clauses (**OLL+WCE+SS/CGSS2, all eqs**).

More study should be done on the impact of the equivalence constraints before certain conclusions can be reached about their usefulness. The question of whether the differences in the results of our novel implementation here compared to results of the PySAT-based variants have to do with differences in implementation or in language or due to some other reason is outside the scope of this thesis. It is also possible that more suitable value choices could provide further improvements for the C++ implementation.

6 Conclusions

This thesis focused on practical algorithms for solving NP-hard real-world optimization problems. In particular, we proposed improvements to so-called core-guided algorithms for the declarative optimization paradigm of MaxSAT. The main contribution is a structure-sharing technique aiming at improving the performance of core-guided algorithms for MaxSAT in two ways: by decreasing the size of the cardinality constraints added to the working instance, and by improving the propagation properties of the cardinality constraints thus allowing more effective reasoning over different cores.

We discussed the potential challenges concerning the efficient realization of structure-sharing and proposed heuristics to overcome the difficulties. We proposed a generalized version of the cardinality constraints employed by PMRES to enable more efficient structure-sharing. To select efficiently what to share, we proposed a greedy algorithm. We also proposed a heuristic algorithm for selectively adding equivalence-defining clauses to some parts of the cardinality constraints.

We evaluated empirically the impact of the proposed technique on the performance of different core-guided solvers. The result demonstrated that structure-sharing can decrease the number of variables and the number of clauses in the added constraints and improve runtimes, even though the improvements seem to concentrate on a relatively small number of instances. The results concerning the impact of the equivalence constraints highlighted the need for additional research on the subject.

Concerning possible future work related to the work presented in this thesis, the following three directions would be worth considering.

(1) *Incorporating structure-sharing into other cardinality constraint encodings and other core-guided MaxSAT algorithms.* In the original publication of OLL for MaxSAT [65], three different cardinality constraint encodings were considered: sorting networks [38], sequential counters [77] and totalizers [20]. In a later work by different authors [71] 4-way merge selection networks [48] and direct networks [12] were selected for an efficient implementation of OLL. Concerning other core-guided algorithms than OLL and PMRES, to the best of our understanding, structure-sharing could be incorporated at least to WPM3 [10] and K [2]

(2) *Sharing structures with extra variables.* When we discussed the structure-sharing, we considered sharing subtrees in a way that all input variables in a shared subtree are in the cores for which the shared subtree is constructed. However, allowing extra variables may in some cases enable a smaller total structure without affecting the correctness. Especially, this is worth considering when looking for reusable structures from previously constructed cardinality constraints.

(3) *More in-depth understanding of the propagation properties of implication-defining and equivalence-defining clauses.* The results of the empirical evaluation we run did not give clear results about how useful the equivalence-defining clauses are and how they should be added in order to achieve the best performance (or does it matter at all). More study is needed on this to reach conclusions. The question of the impact equivalence constraints in PMRES might also be worth considering.

(4) *Using structure-sharing in other approaches to MaxSAT solving.* Structure-sharing could be applied in the context of the IHS approach for MaxSAT extended with abstract cores [21]. A form of structure-sharing could also be incorporated into the model-improving approach, but in that context, the use of structure-sharing would be somewhat different, possibly more like the approach proposed for pseudo-Boolean constraints in the context of ASP [27].

A Proofs for generalized PMRES

In this appendix, we study COUNTING-GPMRES cardinality constraint, defined as follows.

Definition A.1.1. *COUNTING-GPMRES defines a set of output variables for a set of input variables κ with semantics $o_\kappa^{S_1,S_2} \leftrightarrow \left(\bigvee_{v_i \in S_1} v_i \wedge \bigvee_{v_j \in S_2} v_j\right)$ where $S_1 \subseteq \kappa$ and $S_2 \subseteq \kappa$, $|S_1| > 0$, $|S_2| > 0$ in a way that satisfies the counting property, i.e., assigning exactly k input variables to 1 enforces exactly $k-1$ output variables to 1. (recall Definition 3.2.1).*

We will show that any way of selecting the subsets S_1 and S_2 of COUNTING-GPMRES forms a structure that can be viewed as a binary tree. In particular, this implies that the GPMRES-ENC encoding introduced in Chapter 3 is expressive enough to capture all ways of selecting the sets S_1 and S_2 to each output variable that satisfy the definition of COUNTING-GPMRES.

In the following, we view $o^{S_1,S_2} \equiv o^{S_2,S_1}$.

Consider now an arbitrary instantiation of COUNTING-GPMRES. The fact that it can be viewed as a binary tree follows from the following results.

Lemma A.1.1. *The number of the output variables of COUNTING-GPMRES for $|\kappa|$ input variables must be $|\kappa| - 1$.*

Proof. Assigning all $|\kappa|$ input variables to 1 forces every output variable to 1 (follows from the semantics of the output variables). In such a situation, the number of output variables assigned to 1 should be $|\kappa| - 1$ (counting property). Thus the total number of output variables must be $|\kappa| - 1$. □

Lemma A.1.2. *Let $o_\kappa^{S_1,S_2}$ be an output variable of COUNTING-GPMRES. There is no variable $v \in \kappa$ for which $v \in S_1$ and $v \in S_2$, i.e., S_1 and S_2 are disjoint.*

Proof. Otherwise assigning only v of κ to 1 would force the output variable $o_\kappa^{S_1,S_2}$ to 1, breaking the counting property. □

Lemma A.1.3. *For any two input variables $x \in \kappa$ and $y \in \kappa$, COUNTING-GPMRES has exactly one output variable $o_\kappa^{S_1,S_2}$, for which $x \in S_1$ and $y \in S_2$*

Proof. Otherwise assigning exactly x and y of κ to 1 will not force exactly one output variable to 1 (which would break the counting property). $\square$

We call the uniquely defined output variable defined in Lemma A.1.3 for a pair (x, y) of input variables the *capturing output variable*. Informally, the capturing output variable will be exactly the one enforced to 1 when x and y are assigned to 1. In the following, we use function $\textsc{co}(x, y)$ to denote the capturing output variable for a pair of variables: $\textsc{co}(x, y) = o^{S_x,S_y}$ where $x \in S_x$ and $y \in S_y$.

Lemma A.1.4. *For any three input variables $x, y, z \in \kappa$, the pairs (x, y), (x, z), (y, z) must have exactly two capturing output variables. Two of the pairs must have the same capturing output variable.*

Proof. Otherwise assigning exactly x, y and z of κ to 1 will not enforce exactly two output variables to 1. $\square$

Lemma A.1.5. *Let $o_\kappa^{S_1,S_2}$ and $o_\kappa^{S_3,S_4}$ be two output variables of* COUNTING-GPMRES. *If there exists $x \in S_1$ and $y \in S_2$ for which $x, y \in S_3$, then $S_1 \cup S_2 \subseteq S_3$.*

Proof. If $S_1 \cup S_2 = \{x, y\}$ the case is trivial. Assume $z \in S_1 \cup S_2$. Without loss of generality, assume $z \in S_1$. We show that $z \in S_3$. Assume $w \in S_4$. Since $\textsc{co}(z, y) \neq \textsc{co}(y, w)$, either $\textsc{co}(z, w) = \textsc{co}(z, y)$ or $\textsc{co}(z, w) = \textsc{co}(y, w)$ (Lemma A.1.4). Since $\textsc{co}(x, w) = o_\kappa^{S_3,S_4}$ and $x \in S_1$, by Lemma A.1.3 $w \notin S_2$ (consider output $o_\kappa^{S_1,S_2}$), implying $\textsc{co}(z, w) \neq \textsc{co}(z, y) = o_\kappa^{S_1,S_2}$. Thus $\textsc{co}(z, w) = \textsc{co}(y, w)$ and $z \in S_3$. $\square$

Proposition A.1.6. *Let $o_\kappa^{S_1,S_2}$ and $o_\kappa^{S_3,S_4}$ be two output variables of* COUNTING-GPMRES. *If $(S_1 \cup S_2) \cap (S_3 \cup S_4)$ is not empty then either $S_1 \cup S_2 \subseteq S_3$, $S_1 \cup S_2 \subseteq S_4$, $S_3 \cup S_4 \subseteq S_1$ or $S_3 \cup S_4 \subseteq S_2$.*

Proof. Let $x \in (S_1 \cup S_2) \cap (S_3 \cup S_4)$. Without loss of generality, assume $x \in S_1$ and $x \in S_3$. Let y be some variable in S_2 and z some variable in S_4. Since $\textsc{co}(x, y) \neq \textsc{co}(x, z)$, either $\textsc{co}(y, z) = \textsc{co}(x, y)$ meaning $z \in S_1$ or $\textsc{co}(y, z) = \textsc{co}(x, z)$ meaning $y \in S_3$.

Assume there exists a variable $y \in S_2$ for which $y \in S_3$. In that case $S_1 \cup S_2 \subseteq S_3$ (Lemma A.1.5). Similarly, if there exists $z \in S_4$ for which $z \in S_1$, then $S_3 \cup S_4 \subseteq S_1$. $\square$

From Proposition A.1.6 it follows that the output variables of COUNTING-GPMRES can be viewed as forming a binary tree like structure as follows. If for two output variables, o^{S_1,S_2} and o^{S_3,S_4}, $(S_1 \cup S_2) \subseteq S_3$ or $(S_1 \cup S_2) \subseteq S_4$, o^{S_1,S_2} is viewed as a descendant of o^{S_3,S_4}. Since there are two sets for each output variable that define the descendants, the forming tree is a binary tree. Notice that for each pair of output variables, o^{S_1,S_2} and o^{S_3,S_4}, either one is a descendant of the other or $S_1 \cup S_2$ and $S_3 \cup S_4$ are distinct. Since there are $|\kappa| - 1$ output variables (Lemma A.1.1) and $|\kappa|$ input variables (leaves), the tree is complete and connected. Thus GPMRES-ENC encoding can express any COUNTING-GPMRES.

B Command line options for solver variants

For reproducability, Table B.1 details the exact command line options needed to run each of the solver variants in our empirical evaluation.

Table B.1: Solver variants and their parameters

Variant name	**Command and parameters**
OLL+WCE+SS/CGSS2	`./cgss2 -p instance.wcnf[.gz]`
OLL+WCE+SS/CGSS2, all eqs	`./cgss2 -p --all-eqs instance.wcnf[.gz]`
OLL+WCE+SS/CGSS2, no eqs	`./cgss2 -p --no-eqs instance.wcnf[.gz]`
OLL+WCE/CGSS2	`./cgss2 -p --no-ss --no-eqs instance.wcnf[.gz]`
OLL/CGSS2	`./cgss2 -p --no-ss --no-wce instance.wcnf[.gz]`
OLL+WCE+SS/CGSS	`python rc2.py -plxamW -T 1,16 -E 50,50 instance.wcnf[.gz]`
OLL+WCE+SS/CGSS, all eqs	`python rc2.py -plxamWQ -T 1,16 instance.wcnf[.gz]`
OLL+WCE+SS/CGSS, no eqs	`python rc2.py -plxamW -T 1,16 instance.wcnf[.gz]`
OLL+WCE/CGSS	`python rc2.py -plxamWn instance.wcnf[.gz]`
OLL/CGSS	`python rc2.py -plxamWnN instance.wcnf[.gz]`
OLL/RC2	`python rc2.py -c b instance.wcnf[.gz]`
PMRES+WCE+SS/CGSS2	`./cgss2 -p --pmres instance.wcnf[.gz]`
PMRES+WCE/CGSS2	`./cgss2 -p --pmres --no-ss instance.wcnf[.gz]`
PMRES/CGSS2	`./cgss2 -p --pmres --no-ss --no-wce instance.wcnf[.gz]`
PMRES+WCE+SS/CGSS	`python rc2.py -plamWP -T 1,16 instance.wcnf[.gz]`
PMRES+WCE/CGSS	`python rc2.py -plamWnP instance.wcnf[.gz]`
PMRES/CGSS	`python rc2.py -plamWnNP instance.wcnf[.gz]`

Bibliography

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. “Generic ILP versus specialized 0-1 ILP: An update”. In: *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, ICCAD 2002, San Jose, California, USA, November 10-14, 2002*. Ed. by L. T. Pileggi and A. Kuehlmann. ACM / IEEE Computer Society, 2002, pp. 450–457. DOI: 10.1145/774572.774638. URL: https://doi.org/10.1145/774572.774638.

[2] M. Alviano, C. Dodaro, and F. Ricca. “A MaxSAT algorithm using cardinality constraints of bounded size”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Q. Yang and M. J. Wooldridge. AAAI Press, 2015, pp. 2677–2683. URL: http://ijcai.org/Abstract/15/379.

[3] B. Andres, B. Kaufmann, O. Matheis, and T. Schaub. “Unsatisfiability-based optimization in clasp”. In: *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*. Ed. by A. Dovier and V. S. Costa. Vol. 17. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 211–221. DOI: 10.4230/LIPIcs.ICLP.2012.211. URL: https://doi.org/10.4230/LIPIcs.ICLP.2012.211.

[4] M. F. Anjos. “Semidefinite optimization approaches for satisfiability and maximum-satisfiability problems”. In: *J. Satisf. Boolean Model. Comput.* 1.1 (2006), pp. 1–47. DOI: 10.3233/sat190001. URL: https://doi.org/10.3233/sat190001.

[5] C. Ansótegui, M. L. Bonet, J. Gabàs, and J. Levy. “Improving SAT-based weighted MaxSAT solvers”. In: *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*. Ed. by M. Milano. Vol. 7514. Lecture Notes in Computer Science. Springer, 2012, pp. 86–101. DOI: 10.1007/978-3-642-33558-7_9. URL: https://doi.org/10.1007/978-3-642-33558-7_9.

[6] C. Ansótegui, M. L. Bonet, J. Gabàs, and J. Levy. “Improving WPM2 for (weighted) partial MaxSAT”. In: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*. Ed. by C. Schulte. Vol. 8124. Lecture Notes in Computer Science. Springer,

2013, pp. 117–132. DOI: 10.1007/978-3-642-40627-0_12. URL: https://doi.org/10.1007/978-3-642-40627-0_12.

[7] C. Ansótegui, M. L. Bonet, and J. Levy. “SAT-based MaxSAT algorithms”. In: *Artif. Intell.* 196 (2013), pp. 77–105. DOI: 10.1016/j.artint.2013.01.002. URL: https://doi.org/10.1016/j.artint.2013.01.002.

[8] C. Ansótegui, M. L. Bonet, and J. Levy. “Solving (weighted) partial MaxSAT through satisfiability testing”. In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings.* Ed. by O. Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 427–440. DOI: 10.1007/978-3-642-02777-2_39. URL: https://doi.org/10.1007/978-3-642-02777-2_39.

[9] C. Ansótegui, F. Didier, and J. Gabàs. “Exploiting the structure of unsatisfiable cores in MaxSAT”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015.* Ed. by Q. Yang and M. J. Wooldridge. AAAI Press, 2015, pp. 283–289. URL: http://ijcai.org/Abstract/15/046.

[10] C. Ansótegui and J. Gabàs. “WPM3: An (in)complete algorithm for weighted partial MaxSAT”. In: *Artif. Intell.* 250 (2017), pp. 37–57. DOI: 10.1016/j.artint.2017.05.003. URL: https://doi.org/10.1016/j.artint.2017.05.003.

[11] S. Arora and B. Barak. “NP and NP completeness”. In: *Computational Complexity - A Modern Approach.* Cambridge University Press, 2009. Chap. 2, pp. 38–67. ISBN: 978-0-521-42426-4. URL: http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264.

[12] R. Asın, R. Nieuwenhuis, A. Oliveras, and E. Rodrıguez-Carbonell. “Cardinality Networks: a theoretical and empirical study”. In: *Constraints An Int. J.* 16.2 (2011), pp. 195–221. DOI: 10.1007/s10601-010-9105-0. URL: https://doi.org/10.1007/s10601-010-9105-0.

[13] 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 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings.* Ed. by M. Järvisalo and A. V. Gelder. Vol. 7962. Lecture Notes in Computer Science. Springer, 2013, pp. 309–317. DOI: 10.1007/978-3-642-39071-5_23. URL: https://doi.org/10.1007/978-3-642-39071-5_23.

[14] F. Avellaneda. “A short description of the solver EvalMaxSAT”. In: *MaxSAT Evaluation 2020: Solver and Benchmark Description.* Department of Computer Science Report Series B B-2020-2 (2020). Ed. by F. Bacchus, J. Berg, M. Järvisalo, and R. Martins, pp. 8–9.

[15] F. Bacchus, J. Berg, M. Järvisalo, and R. Martins, eds. *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions.* Vol. B-2020-2. Department of Computer Science Series of Publications B. University of Helsinki, 2020.

[16] F. Bacchus, M. Järvisalo, and R. Martins. “Maximum satisfiability”. In: *Handbook of Satisfiability - Second Edition.* Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 929–991. DOI: 10.3233/FAIA201008. URL: https://doi.org/10.3233/FAIA201008;%20https://dblp.org/rec/series/faia/BacchusJM21.bib.

[17] F. Bacchus, M. Järvisalo, and R. Martins. “MaxSAT Evaluation 2018: New developments and detailed results”. In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 99–131. DOI: 10.3233/SAT190119. URL: https://doi.org/10.3233/SAT190119.

[18] F. Bacchus, M. Järvisalo, and R. Martins, eds. *MaxSAT Evaluation 2019: Solver and Benchmark Descriptions.* Vol. B-2019-2. Department of Computer Science Series of Publications B. University of Helsinki, 2019.

[19] F. Bacchus and N. Narodytska. “Cores in core based MaxSat algorithms: An analysis”. In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings.* Ed. by C. Sinz and U. Egly. Vol. 8561. Lecture Notes in Computer Science. Springer, 2014, pp. 7–15. DOI: 10.1007/978-3-319-09284-3_2. URL: https://doi.org/10.1007/978-3-319-09284-3%5C_2.

[20] O. Bailleux and Y. Boufkhad. “Efficient CNF encoding of Boolean cardinality constraints”. In: *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings.* Ed. by F. Rossi. Vol. 2833. Lecture Notes in Computer Science. Springer, 2003, pp. 108–122. DOI: 10.1007/978-3-540-45193-8_8. URL: https://doi.org/10.1007/978-3-540-45193-8_8.

[21] J. Berg, F. Bacchus, and A. Poole. “Abstract cores in implicit hitting set MaxSat solving”. In: *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings.* Ed. by L.

Pulina and M. Seidl. Vol. 12178. Lecture Notes in Computer Science. Springer, 2020, pp. 277–294. DOI: 10.1007/978-3-030-51825-7_20. URL: https://doi.org/10.1007/978-3-030-51825-7_20.

[22] J. Berg, E. Demirovic, and P. J. Stuckey. “Core-boosted linear search for incomplete MaxSAT”. In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*. Ed. by L.-M. Rousseau and K. Stergiou. Vol. 11494. Lecture Notes in Computer Science. Springer, 2019, pp. 39–56. DOI: 10.1007/978-3-030-19212-9_3. URL: https://doi.org/10.1007/978-3-030-19212-9_3.

[23] J. Berg and M. Järvisalo. “Weight-aware core extraction in SAT-based MaxSAT solving”. In: *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Ed. by J. C. Beck. Vol. 10416. Lecture Notes in Computer Science. Springer, 2017, pp. 652–670. DOI: 10.1007/978-3-319-66158-2_42. URL: https://doi.org/10.1007/978-3-319-66158-2_42.

[24] D. L. Berre and A. Parrain. “The Sat4j library, release 2.2”. In: *J. Satisf. Boolean Model. Comput.* 7.2-3 (2010), pp. 59–6. DOI: 10.3233/sat190075. URL: https://doi.org/10.3233/sat190075.

[25] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020”. In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[26] N. S. Bjørner and N. Narodytska. “Maximum satisfiability using cores and correction sets”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Q. Yang and M. J. Wooldridge. AAAI Press, 2015, pp. 246–252. URL: http://ijcai.org/Abstract/15/041.

[27] J. Bomanson, M. Gebser, and T. Janhunen. “Improving the normalization of weight rules in answer set programs”. In: *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*. Ed. by E. Fermé and J. Leite. Vol. 8761. Lecture Notes in Computer Sci-

ence. Springer, 2014, pp. 166–180. DOI: 10.1007/978-3-319-11558-0_12. URL: https://doi.org/10.1007/978-3-319-11558-0_12.

[28] B. Borchers and J. Furman. “A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems”. In: *J. Comb. Optim.* 2.4 (1998), pp. 299–306. DOI: 10.1023/A:1009725216438. URL: https://doi.org/10.1023/A:1009725216438.

[29] E. Boros and P. L. Hammer. “Pseudo-Boolean optimization”. In: *Discret. Appl. Math.* 123.1-3 (2002), pp. 155–225. DOI: 10.1016/S0166-218X(01)00341-9. URL: https://doi.org/10.1016/S0166-218X(01)00341-9.

[30] S. Cai and Z. Lei. “Old techniques in new ways: Clause weighting, unit propagation and hybridization for maximum satisfiability”. In: *Artif. Intell.* 287 (2020), p. 103354. DOI: 10.1016/j.artint.2020.103354. URL: https://doi.org/10.1016/j.artint.2020.103354.

[31] M. Conforti, G. Cornuéjols, G. Zambelli, et al. *Integer Programming.* Springer, 2014.

[32] S. A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. Ed. by M. A. Harrison, R. B. Banerji, and J. D. Ullman. ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[33] J. Davies. “Solving MAXSAT by Decoupling Optimization and Satisfaction”. PhD thesis. University of Toronto, Canada, 2014. URL: http://hdl.handle.net/1807/43539.

[34] J. Davies and F. Bacchus. “Solving MAXSAT by solving a sequence of simpler SAT instances”. In: *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings.* Ed. by J. H.-M. Lee. Vol. 6876. Lecture Notes in Computer Science. Springer, 2011, pp. 225–239. DOI: 10.1007/978-3-642-23786-7_19. URL: https://doi.org/10.1007/978-3-642-23786-7_19.

[35] E. Demirovic and P. J. Stuckey. “Techniques inspired by local search for incomplete MaxSAT and the linear algorithm: Varying resolution and solution-guided search”. In: *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings.* Ed. by T. Schiex and S. de Givry. Vol. 11802. Lecture Notes in Computer Sci-

ence. Springer, 2019, pp. 177–194. DOI: 10.1007/978-3-030-30048-7_11. URL: https://doi.org/10.1007/978-3-030-30048-7_11.

[36] N. Eén and N. Sörensson. "An extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. Ed. by E. Giunchiglia and A. Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37. URL: https://doi.org/10.1007/978-3-540-24605-3_37.

[37] N. Eén and N. Sörensson. "Temporal induction by incremental SAT solving". In: *Electron. Notes Theor. Comput. Sci.* 89.4 (2003), pp. 543–560. DOI: 10.1016/S1571-0661(05)82542-3. URL: https://doi.org/10.1016/S1571-0661(05)82542-3.

[38] N. Eén and N. Sörensson. "Translating pseudo-Boolean constraints into SAT". In: *J. Satisf. Boolean Model. Comput.* 2.1-4 (2006), pp. 1–26. DOI: 10.3233/sat190014. URL: https://doi.org/10.3233/sat190014.

[39] Z. Fu and S. Malik. "On solving the partial MAX-SAT problem". In: *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*. Ed. by A. Biere and C. P. Gomes. Vol. 4121. Lecture Notes in Computer Science. Springer, 2006, pp. 252–265. DOI: 10.1007/11814948_25. URL: https://doi.org/10.1007/11814948_25.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[41] M. Gelfond and V. Lifschitz. "Classical negation in logic programs and disjunctive databases". In: *New Gener. Comput.* 9.3/4 (1991), pp. 365–386. DOI: 10.1007/BF03037169. URL: https://doi.org/10.1007/BF03037169.

[42] P. Hansen and B. Jaumard. "Algorithms for the maximum satisfiability problem". In: *Computing* 44.4 (1990), pp. 279–303. DOI: 10.1007/BF02241270. URL: https://doi.org/10.1007/BF02241270.

[43] F. Heras, J. Larrosa, and A. Oliveras. "MiniMaxSat: A new weighted Max-SAT solver". In: *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*. Ed. by J. Marques-Silva and K. A. Sakallah. Vol. 4501. Lecture Notes in Computer Science. Springer, 2007, pp. 41–55. DOI: 10.1007/978-3-540-72788-0_8. URL: https://doi.org/10.1007/978-3-540-72788-0_8.

[44] A. Ignatiev, A. Morgado, and J. Marques-Silva. "PySAT: A Python toolkit for prototyping with SAT oracles". In: *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by O. Beyersdorff and C. M. Wintersteiger. Vol. 10929. Lecture Notes in Computer Science. Springer, 2018, pp. 428–437. DOI: 10.1007/978-3-319-94144-8_26. URL: https://doi.org/10.1007/978-3-319-94144-8_26.

[45] A. Ignatiev, A. Morgado, and J. Marques-Silva. "RC2: An efficient MaxSAT solver". In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 53–64. DOI: 10.3233/SAT190116. URL: https://doi.org/10.3233/SAT190116.

[46] H. Ihalainen, J. Berg, and M. Järvisalo. "Refined core relaxation for core-guided MaxSAT solving". In: *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*. Ed. by L. D. Michel. Vol. 210. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 28:1–28:19. DOI: 10.4230/LIPIcs.CP.2021.28. URL: https://doi.org/10.4230/LIPIcs.CP.2021.28.

[47] S. Joshi, P. Kumar, S. Rao, and R. Martins. "Open-WBO-Inc: approximation strategies for incomplete weighted MaxSAT". In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 73–97. DOI: 10.3233/SAT190118. URL: https://doi.org/10.3233/SAT190118.

[48] M. Karpinski and M. Piotrów. "Encoding cardinality constraints using multiway merge selection networks". In: *Constraints An Int. J.* 24.3-4 (2019), pp. 234–251. DOI: 10.1007/s10601-019-09302-0. URL: https://doi.org/10.1007/s10601-019-09302-0.

[49] Z. Lei and S. Cai. "Solving (weighted) partial MaxSAT by dynamic local search for SAT". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by J. Lang. ijcai.org, 2018, pp. 1346–1352. DOI: 10.24963/ijcai.2018/187. URL: https://doi.org/10.24963/ijcai.2018/187.

[50] L. A. Levin. "Universal sequential search problems". In: *Problemy peredachi informatsii* 9.3 (1973), pp. 115–116.

[51] C. M. Li and F. Manyà. "MaxSAT, hard and soft constraints". In: *Handbook of Satisfiability - Second Edition*. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 903–927. DOI: 10.3233/FAIA201007. URL: https://doi.org/10.3233/FAIA201007;%20https://dblp.org/rec/series/faia/LiM21.bib.

[52] C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. "Resolution-based lower bounds in MaxSAT". In: *Constraints An Int. J.* 15.4 (2010), pp. 456–484. DOI: 10.1007/s10601-010-9097-9. URL: https://doi.org/10.1007/s10601-010-9097-9.

[53] C.-M. Li, Z. Xu, J. Coll, F. Manyà, D. Habet, and K. He. "Boosting branch-and-bound MaxSAT solvers with clause learning". In: *AI Commun.* 35.2 (2022), pp. 131–151. DOI: 10.3233/AIC-210178. URL: https://doi.org/10.3233/AIC-210178.

[54] C.-M. Li, Z. Xu, J. Coll, F. Manyà, D. Habet, and K. He. "Combining clause learning and branch and bound for MaxSAT". In: *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*. Ed. by L. D. Michel. Vol. 210. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 38:1–38:18. DOI: 10.4230/LIPIcs.CP.2021.38. URL: https://doi.org/10.4230/LIPIcs.CP.2021.38.

[55] V. M. Manquinho, J. P. Marques-Silva, and J. Planes. "Algorithms for weighted Boolean optimization". In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Ed. by O. Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 495–508. DOI: 10.1007/978-3-642-02777-2_45. URL: https://doi.org/10.1007/978-3-642-02777-2_45.

[56] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce. "Boolean lexicographic optimization: algorithms & applications". In: *Ann. Math. Artif. Intell.* 62.3-4 (2011), pp. 317–343. DOI: 10.1007/s10472-011-9233-2. URL: https://doi.org/10.1007/s10472-011-9233-2.

[57] J. Marques-Silva, I. Lynce, and S. Malik. "Conflict-driven clause learning SAT solvers". In: *Handbook of Satisfiability - Second Edition*. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 133–182. DOI: 10.3233/FAIA200987. URL: https://doi.org/10.3233/FAIA200987.

[58] J. Marques-Silva and J. Planes. "Algorithms for maximum satisfiability using unsatisfiable cores". In: *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*. Ed. by D. Sciuto. ACM, 2008, pp. 408–413. DOI: 10.1109/DATE.2008.4484715. URL: https://doi.org/10.1109/DATE.2008.4484715.

[59] J. Marques-Silva and J. Planes. "On using unsatisfiability for solving maximum satisfiability". In: *CoRR* abs/0712.1097 (2007). arXiv: 0712.1097. URL: http://arxiv.org/abs/0712.1097.

[60] J. P. Marques-Silva. "Minimal unsatisfiability: Models, algorithms and applications (invited paper)". In: *40th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2010, Barcelona, Spain, 26-28 May 2010*. IEEE Computer Society, 2010, pp. 9–14. DOI: 10.1109/ISMVL.2010.11. URL: https://doi.org/10.1109/ISMVL.2010.11.

[61] J. P. Marques-Silva and K. A. Sakallah. "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Trans. Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433. URL: https://doi.org/10.1109/12.769433.

[62] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce. "Incremental cardinality constraints for MaxSAT". In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Ed. by B. O'Sullivan. Vol. 8656. Lecture Notes in Computer Science. Springer, 2014, pp. 531–548. DOI: 10.1007/978-3-319-10428-7_39. URL: https://doi.org/10.1007/978-3-319-10428-7_39.

[63] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce. "On using incremental encodings in unsatisfiability-based MaxSAT solving". In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 59–81. DOI: 10.3233/sat190102. URL: https://doi.org/10.3233/sat190102.

[64] R. Martins, V. M. Manquinho, and I. Lynce. "Open-WBO: A modular MaxSAT solver," in: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by C. Sinz and U. Egly. Vol. 8561. Lecture Notes in Computer Science. Springer, 2014, pp. 438–445. DOI: 10.1007/978-3-319-09284-3_33. URL: https://doi.org/10.1007/978-3-319-09284-3_33.

[65] A. Morgado, C. Dodaro, and J. Marques-Silva. "Core-guided MaxSAT with soft cardinality constraints". In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Ed. by B. O'Sullivan. Vol. 8656. Lecture Notes in Computer Science. Springer, 2014, pp. 564–573. DOI: 10.1007/978-3-319-10428-7_41. URL: https://doi.org/10.1007/978-3-319-10428-7_41.

[66] A. Morgado, F. Heras, M. H. Liffiton, J. Planes, and J. Marques-Silva. "Iterative and core-guided MaxSAT solving: A survey and assessment". In: *Constraints An Int. J.* 18.4 (2013), pp. 478–534. DOI: 10.1007/s10601-013-9146-2. URL: https://doi.org/10.1007/s10601-013-9146-2.

[67] A. Morgado, F. Heras, and J. Marques-Silva. "Improvements to core-guided binary search for MaxSAT". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by A. Cimatti and R. Sebastiani. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 284–297. DOI: 10.1007/978-3-642-31612-8_22. URL: https://doi.org/10.1007/978-3-642-31612-8%5C_22.

[68] A. Morgado, A. Ignatiev, and J. Marques-Silva. "MSCG: Robust core-guided MaxSAT solving". In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 129–134. DOI: 10.3233/sat190105. URL: https://doi.org/10.3233/sat190105.

[69] N. Narodytska and F. Bacchus. "Maximum satisfiability using core-guided MaxSAT resolution". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by C. E. Brodley and P. Stone. AAAI Press, 2014, pp. 2717–2723. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8513.

[70] R. Nieuwenhuis and A. Oliveras. "On SAT modulo theories and optimization problems". In: *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*. Ed. by A. Biere and C. P. Gomes. Vol. 4121. Lecture Notes in Computer Science. Springer, 2006, pp. 156–169. DOI: 10.1007/11814948_18. URL: https://doi.org/10.1007/11814948_18.

[71] M. Piotrów. "UWrMaxSat: Efficient solver for MaxSAT and pseudo-Boolean problems". In: *32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2020, Baltimore, MD, USA, November 9-11, 2020*. IEEE, 2020, pp. 132–

136. DOI: 10.1109/ICTAI50040.2020.00031. URL: https://doi.org/10.1109/ICTAI50040.2020.00031.

[72] J. Planes. "Improved branch and bound algorithms for Max-2-SAT and weighted Max-2-SAT". In: *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*. Ed. by F. Rossi. Vol. 2833. Lecture Notes in Computer Science. Springer, 2003, p. 991. DOI: 10.1007/978-3-540-45193-8_115. URL: https://doi.org/10.1007/978-3-540-45193-8_115.

[73] S. D. Prestwich. "CNF encodings". In: *Handbook of Satisfiability - Second Edition*. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 75–100. DOI: 10.3233/FAIA200985. URL: https://doi.org/10.3233/FAIA200985.

[74] F. Rossi, P. van Beek, and T. Walsh, eds. *Handbook of constraint programming*. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006. ISBN: 978-0-444-52726-4. URL: https://www.sciencedirect.com/science/bookseries/15746526/2.

[75] P. Saikko. "Implicit Hitting Set Algorithms for Constraint Optimization". eng. PhD thesis. 2019.

[76] P. Saikko, J. Berg, and M. Järvisalo. "LMHS: A SAT-IP hybrid MaxSAT solver". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by N. Creignou and D. L. Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 539–546. DOI: 10.1007/978-3-319-40970-2_34. URL: https://doi.org/10.1007/978-3-319-40970-2_34.

[77] C. Sinz. "Towards an optimal CNF encoding of Boolean cardinality constraints". In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*. Ed. by P. van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, 2005, pp. 827–831. DOI: 10.1007/11564751_73. URL: https://doi.org/10.1007/11564751_73.

[78] P. Smirnov, J. Berg, and M. Järvisalo. "Improvements to the implicit hitting set approach to pseudo-Boolean optimization". In: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*. Ed. by K. S. Meel and O. Strichman. Vol. 236. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 13:1–13:18. DOI: 10.4230/LIPIcs.SAT.2022.13. URL: https://doi.org/10.4230/LIPIcs.SAT.2022.13.

[79] G. S. Tseitin. “On the complexity of derivation in propositional calculus”. In: *Automation of Reasoning: Classical papers in Computational Logic*. Springer-Verlag, 1983, 2:466–483.

[80] R. J. Wallace and E. C. Freuder. “Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems”. In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by D. S. Johnson and M. A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1993, pp. 587–615. DOI: 10.1090/dimacs/026/28. URL: https://doi.org/10.1090/dimacs/026/28.

[81] L. A. Wolsey. *Integer Programming*. John Wiley & Sons, 2020.