Re-implementing and Extending a Hybrid SAT–IP Approach to Maximum Satisfiability

Paul Saikko

M. Sc. Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, November 18, 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Paul Saikko

Työn nimi — Arbetets titel — Title

Re-implementing and Extending a Hybrid SAT–IP Approach to Maximum Satisfiability

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M. Sc. Thesis	November 18, 2015	69

Tiivistelmä — Referat — Abstract

Real-world optimization problems, such as those found in logistics and bioinformatics, are often NP-hard. Maximum satisfiability (MaxSAT) provides a framework within which many such problems can be efficiently represented. MaxHS is a recent exact algorithm for MaxSAT. It is a hybrid approach that uses a SAT solver to compute unsatisfiable cores and an integer programming (IP) solver to compute minimum-cost hitting sets for the found cores. This thesis analyzes and extends the MaxHS algorithm. To enable this, the algorithm is re-implemented from scratch using the C++ programming language. The resulting MaxSAT solver LMHS recently gained top positions at an international evaluation of MaxSAT solvers.

This work looks into various aspects of the MaxHS algorithm and its applications. The impact of different IP solvers on the MaxHS algorithm and the behavior induced by different strategies of postponing IP solver calls is examined. New methods of enhancing the computation of unsatisfiable cores in MaxHS are examined. Fast core extraction through parallelization by partitioning soft clauses is explored. A modification of the final conflict analysis procedure of a SAT solver is used to generate additional cores without additional SAT solver invocations. The use of additional constraint propagation procedures in the SAT solver used by MaxHS is investigated. As a case study, acyclicity constraint propagation is implemented and its effectiveness for bounded treewidth Bayesian network structure learning using MaxSAT is evaluated. The extension of MaxHS to the labeled MaxSAT framework, which allows for more efficient use of preprocessing techniques and group MaxSAT encodings in MaxHS, is discussed. The re-implementation of the MaxHS algorithm, LMHS, also enables incrementality in efficiently adding constraints to a MaxSAT instance during the solving process. As a case study, this incrementality is used in solving subproblems with MaxSAT within GOBNILP, a tool for finding optimal Bayesian network structures.

Avainsanat — Nyckelord — Keywords
Discrete Optimization, Maximum Satisfiability, Bayesian Network Structure Learning

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Computationally difficult decision and optimization problems have important real-world uses. For example, the ability to solve problems in hardware and software verification [103, 71] is important for applications such as ensuring the safety of complex, critical systems such as operating system components and modern airplanes. Even more challenging are optimization problems which require us to find, with respect to a given objective function, a maximum or minimum among all possible solutions. Such problems are commonplace in domains such as planning and scheduling, where the difference between an optimal solution and a sub-optimal one can at times be measured millions of dollars. These practical problems include determining the locations of production and storage facilities [69] and facility layout optimization [9]. Similar problems arise in routing air traffic [12] and scheduling course times in universities [35] or shifts in workplaces [73].

Constrained optimization is a convenient way of representing and solving optimization problems. Such problems typically involve optimizing a relatively simple objective function subject to some set of constraints or conditions specified using some mathematical constraint modeling language. This allows for a problem to be built up incrementally and for adding additional constraints as a problem evolves. Many paradigms for representing constrained optimization problems exist. For example, integer programming (IP) [101], answer set programming (ASP) [94, 51], and maximum satisfiability (MaxSAT) [74] all provide ways of representing constrained optimization problems. Each provides a well-defined way of representing constraints. Efficient ways of encoding problems and conditions in these languages are of both practical and theoretical interest. Instead of developing a separate algorithm for every problem domain, algorithms can be developed for these languages. In short, the declarative approach of encoding a problem as a set of constraints allows us to develop more general algorithmic solutions which can be applied to a wide range of problem domains.

Advances in algorithms and solvers for Boolean satisfiability (SAT) have enabled their practical use in solving more general problems such as maximum satisfiability, satisfiability modulo theories (SMT) [95, 102], ASP, and counterexample-guided abstraction refinement (CEGAR) [29]. This thesis focuses on MaxSAT, an optimization version of SAT. MaxSAT is a relatively low-level approach to constrained optimization with its simple language based on conjunctive normal form (CNF) propositional formulas: constraints in MaxSAT (clauses) are simply disjunctions of binary variables or their negations. As a constrained optimization paradigm, MaxSAT is interesting for a number of reasons. Considerable work has been done to develop efficient SAT encodings of various constraints, which can be utilized for MaxSAT. New developments in SAT solving are also applicable to MaxSAT since most MaxSAT solvers solve a sequence of SAT problems. The weighted

partial generalization of MaxSAT allows for a wider range of problems to be conveniently encoded. It allows for some (hard) constraints to be specified as mandatory and allows weights to be associated with other (soft) constraints. This gives a natural way of specifying the relative importance of constraints. MaxSAT (and especially weighted partial MaxSAT) has recently been successfully applied to a number of problems including hardware debugging [62, 106], data analysis [18, 26], model-based diagnosis [79], and bioinformatics [100, 19, 53].

MaxSAT algorithms [89, 3, 54, 70, 36, 92, 24, 87] commonly use a SAT solver as a black box to solve a sequence of SAT problems. Many of these are so-called core-based algorithms [84]. These use a SAT solver to identify sets of constraints which cannot be simultaneously satisfied, called cores, and employ various techniques to optimally account for each one. We focus on MaxHS [37, 38, 39, 36], a recent MaxSAT algorithm which has been shown to be very effective for solving instances in some problem categories. MaxHS can be seen as an instantiation of a general implicit hitting set approach to optimization [88] for MaxSAT. Unlike most MaxSAT algorithms, MaxHS is a "hybrid" approach that utilizes both an IP solver and a SAT solver. This approach sidesteps issues that can arise from the incrementally expanding SAT formula of most MaxSAT solvers. Instead, MaxHS works with a fixed SAT formula and a growing IP problem.

In this work we analyze and extend MaxHS using our from-scratch reimplementation of the algorithm, LMHS. We analyze the effect of a range of so-called non-optimal hitting set strategies for the MaxHS algorithm. Our experiments with extending MaxHS include

- modifying the final conflict analysis procedure of the underlying SAT solver to enable polynomial-time computation of additional cores,
- an exploration of parallelizing the MaxHS algorithm by means of heuristic randomization and partitioning,
- applying external constraint propagation within a MaxSAT solver for constraints that are potentially inefficient to express in CNF, and
- evaluating recent work [20, 21] on integrating SAT-based preprocessing with MaxSAT solvers.

We also develop an incremental MaxSAT API for LMHS and examine an application of incremental MaxSAT solving [100].

This thesis is organized as follows. Section 2 establishes the required prerequisite knowledge of SAT and MaxSAT. Section 3 covers the MaxHS algorithm. Section 4 explains the implementation of our solver LMHS, evaluates its performance with different SAT and IP solvers and non-optimal hitting set strategies, and compares its performance to other state-of-the-art MaxSAT solvers. Section 5 introduces and evaluates our extensions to the

MaxHS algorithm. Sections 5.6 and 6 cover the LMHS incremental API and an application of incremental MaxSAT, respectively. Section 7 concludes this work and outlines possible further work on the topic.

2 Preliminaries

This section covers prerequisite information. We begin with a short overview of satisfiability and satisfiability solvers. We then review an optimization version of satisfiability, maximum satisfiability, and its common extensions. We also survey some recent and classical methods for solving maximum satisfiability problems. This section ends with an overview of another discrete optimization paradigm, integer programming, which plays an important role in the MaxHS algorithm.

2.1 SAT

Boolean satisfiability (SAT) [47] is one of the most studied problems in computational complexity theory. Given a propositional logic formula $\mathcal{F}$ over a set of Boolean variables X, it asks if there exists a truth assignment $\tau : X \to \{0, 1\}$ such that $\mathcal{F}$ is true under τ. If such an assignment exists, the formula $\mathcal{F}$ is *satisfiable*. Otherwise $\mathcal{F}$ is *unsatisfiable*. With the exception of restricted classes of formulas such as 2-SAT [7] and Horn-SAT [40], SAT is an NP-complete problem [31]. In other words, no polynomial-time algorithm is known for the problem, but solutions can be verified in polynomial time. Given a complete truth assignment τ for a formula $\mathcal{F}$, the time complexity of determining $\tau(\mathcal{F})$ is linear in the size of $\mathcal{F}$.

Boolean satisfiability is often restricted to conjunctive normal form (CNF) formulas. CNF is a subset of propositional logic restricted to conjunctions ($\wedge$) of disjunctions ($\vee$) of Boolean variables x and their negations $\neg x$.

Definition 1. Syntax of CNF.

- A *literal* l is a Boolean variable x or its negation $\neg x$.
- A *clause* C is a disjunction $\bigvee_{i=1}^{m} l_i$ of literals $l_1, \ldots, l_m$.
- A *formula* in CNF is a conjunction of clauses $\bigwedge_{i=1}^{n} C_i$.

Any propositional logic formula can be converted to CNF using the rules of Boolean algebra [98]. Such a conversion will result in an exponential number of clauses in the worst case, rendering it impractical for general use. However, more efficient encodings exist. For example, the commonly used Tseitin encoding [105] introduces only a linear number of new clauses in the conversion, at the cost of a linear number of new variables.

When convenient, we treat a CNF formula as a set of clauses, trusting that their implicit conjunction is clear from the context. The advantage of

CNF is that its simple clausal structure makes it convenient to process—and reason about—large formulas. Indeed, the Boolean satisfiability problem for CNF formulas (CNF-SAT) is so common that it is usually simply referred to as SAT. Definition 2 formalizes the concept of satisfiability for CNF formulas.

Definition 2. Semantics of CNF-SAT.

- A (complete) truth assignment τ for the variables X of a formula $\mathcal{F}$ is a function $\tau : X \to \{0, 1\}$, which assigns either $x = 0$ or $x = 1$ for every $x \in X$.
- The literal x is true if $x = 1$ under τ, and the literal $\neg x$ is true if $x = 0$ under τ. If a literal l is true under τ, then $\tau(l) = 1$. If the variable of a literal l is not assigned under τ, it is undefined.
- A clause $C = \bigvee_{i=1}^{m} l_i$ is true under (or equivalently, satisfied by) τ, $\tau(C) = 1$, if and only if at least one of the literals l_i is true under τ. An empty clause is unsatisfiable.
- A formula $\mathcal{F} = \bigwedge_{i=1}^{n} C_i$ is satisfied by τ, $\tau(\mathcal{F}) = 1$, if and only if *every* clause C_i is true under τ. If $\tau(\mathcal{F}) = 1$, then τ is a satisfying assignment for $\mathcal{F}$. An empty formula is satisfiable.

2.2 SAT solvers

Despite the computational complexity of SAT, developments in solving techniques, driven in part by frequent competitions [60], have led to very efficient SAT solvers and algorithms. Many of these solvers are based on conflict-driven clause learning (CDCL) [82] algorithm, the core parts of which were first introduced in [85]. Intuitively, CDCL is able to implicitly exploit the structure present in real-world instances. While the CDCL algorithm is not well-suited to randomly generated instances due to the lack of real-world structure, it has become practical to use a SAT solver for large formulas in many applications [78]. These developments have also opened the door for applying these solvers as NP oracles for optimization problems beyond NP such as AI planning [67], QBF solving [59], and propositional circumscription [58].

Algorithm 1 gives a sketch of the CDCL algorithm. The algorithm begins by applying constraint propagation. This is commonly implemented as *unit propagation* (we consider additional propagation techniques based on external constraints in Section 5.3). Given a partial truth assignment τ, a clause $C = \bigvee_{i=1}^{m} l_i$, is *unit under* τ if there exists a literal $l_i \in C$ such that $\tau(l_j)$ is undefined and $\tau(l_i) = 0$ for all $i \neq j$. A unit clause is trivially unit under τ if its single literal has an undefined value. If a clause is unit under τ, unit propagation will extend τ with a variable assignment which satisfies the undefined literal. This new assignment can then be propagated in the same

Algorithm 1 Conflict-driven clause learning

```
1: function CDCL(F)
2:     D ← ∅
3:     τ ← ∅                                   ▷ initialize an empty assignment
4:     while True do
5:         (c, τ) ← Propagate(A, τ, F)
6:         if c contains a conflict then
7:             if D = ∅ then
8:                 return Unsatisfiable
9:             else                                   ▷ learn a conflict clause
10:                C ← AnalyzeConflict(c)
11:                F ← F ∪ {C}
12:                (D, τ) ← UndoDecisions(D, τ, C)          ▷ backjump
13:        else if τ(F) = 1 then
14:            return Satisfiable, τ
15:        else                                    ▷ make a new decision
16:            Choose a variable x unassigned by τ
17:            Make a decision for x: d ← {x = 0} or d ← {x = 1}
18:            D ← D ∪ {d}
```

manner. The Propagate function of Algorithm 1 returns a description of the conflict c in terms of conflicting assignments if propagation results in an unsatisfied clause. Otherwise a (possibly partial) assignment τ is given.

If the variable assignment given by propagation satisfies all clauses, then $\mathcal{F}$ is satisfiable. If a partial assignment does not cause a conflict or satisfy all clauses, a new decision is made. A currently unassigned variable x and an assignment $\{x = 0\}$ or $\{x = 1\}$ is chosen, and the algorithm continues by propagating this assignment.

One of the features which differentiates CDCL from a standard backtracking search is how conflicts are handled. Rather than undoing a single decision and continuing search, the conflict is analyzed to yield a *conflict clause* C, which captures the cause of unsatisfiability in the current assignment. The clause C can then be added to $\mathcal{F}$ and search can *backjump*, eliminating a potentially large portion of the search space. Backjumping is accomplished by undoing decisions which caused the conflict. In Algorithm 1, the conflict clause is computed by the AnalyzeConflict procedure. Decisions and any assignments propagated from them are undone based on the conflict clause by UndoDecisions.

Unlike a traditional backtracking search, CDCL does not terminate for an unsatisfiable formula only after considering the entire search space. Rather, the conflict analysis procedure of CDCL will eventually learn an empty clause, at which point $\mathcal{F}$ must be unsatisfiable. The CDCL algorithm is

often improved in practice by heuristics for choosing decision variables (e.g. VSIDS [90]), search restarts [56], and other extensions [66].

2.3 Maximum satisfiability

Maximum satisfiability (MaxSAT) is an optimization extension of Boolean satisfiability [74]. In this section we review MaxSAT and its extensions, as well as the relevant notation and terminology.

Unweighted MaxSAT Given an unsatisfiable CNF formula $\mathcal{F}$, it is natural to ask what is the largest number of clauses in $\mathcal{F}$ that can be simultaneously satisfied. The MaxSAT problem asks us to find an assignment τ that maximizes the number of satisfied clauses $\sum_{C \in \mathcal{F}} \tau(C)$. Equivalently, the task is to find a subformula $\mathcal{F}' \subset \mathcal{F}$ with a maximum number of clauses such that there exists a truth assignment τ for which $\tau(\mathcal{F}') = 1$. The set of all such *optimal* solutions for a formula $\mathcal{F}$ is $opt(\mathcal{F})$. The number of clauses not satisfied by a truth assignment τ is $cost(\tau) = \sum_{C \in \mathcal{F}} (1 - \tau(C))$. If $\mathcal{F}$ is satisfiable, then there exists τ such that $\sum_{C \in \mathcal{F}} \tau(C) = |\mathcal{F}|$, and $cost(\tau) = 0$.

As an example, consider the following unsatisfiable CNF formula, and the possible assignments for its variables x_1 and x_2.

$$F = (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1) \wedge (x_2). \quad (1)$$

The truth assignment $\{x_1 = 0, x_2 = 0\}$ satisfies three clauses, $\{x_1 = 1, x_2 = 0\}$ and $\{x_1 = 0, x_2 = 1\}$ both satisfy four clauses, while $\{x_1 = 1, x_2 = 1\}$ satisfies five clauses. Thus, $\{x_1 = 1, x_2 = 1\} \in opt(\mathcal{F})$. This assignment τ satisfies five of the six clauses, so $cost(\tau) = 1$ due to the fact that τ does not satisfy the clause $(\neg x_1 \vee \neg x_2)$.

Partial MaxSAT There are several extensions to MaxSAT which allow for more natural or succinct encodings of problems. A common extension is partial MaxSAT. An instance of the partial MaxSAT problem consists of a set of *hard* clauses $\mathcal{F}_h$, which must be satisfied, and a set of *soft* clauses $\mathcal{F}_s$, which need not be satisfied. Here we make the distinction between a partial MaxSAT *instance* $(\mathcal{F}_h, \mathcal{F}_s)$ and the CNF *formulas* $\mathcal{F}_h$ and $\mathcal{F}_s$. A partial MaxSAT problem asks us to find a truth assignment $\hat{\tau}$ such that $\hat{\tau}(\mathcal{F}_h) = 1$ and $\hat{\tau} \in \arg\max_\tau \sum_{C \in \mathcal{F}_s} \tau(C)$. Similar to plain MaxSAT, the cost of a solution to a partial MaxSAT instance is $cost(\tau) = \sum_{C \in \mathcal{F}_s} (1 - \tau(C))$. If $\mathcal{F}_h$ is unsatisfiable (i.e., there exists no τ for which $\tau(\mathcal{F}_h) = 1$), then the partial MaxSAT instance $(\mathcal{F}_h, \mathcal{F}_s)$ is unsatisfiable regardless of $\mathcal{F}_s$. MaxSAT is a special case of partial MaxSAT, where $\mathcal{F}_h = \emptyset$. A partial MaxSAT instance can be converted into a MaxSAT instance by creating $|\mathcal{F}_s|$ additional copies of each hard clause, and rejecting solutions with a cost greater than $|\mathcal{F}_s|$.

We look again at the example of Equation 1, but now partition it into soft and hard clauses to create the partial MaxSAT instance $(\mathcal{F}_h, \mathcal{F}_s)$:

$$\begin{aligned} \mathcal{F}_h &= (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2), \\ \mathcal{F}_s &= (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1) \wedge (x_2). \end{aligned} \tag{2}$$

Notice that the optimal solution for Equation 1, $\{x_1 = 1, x_2 = 1\}$, is no longer valid as it falsifies the hard clause $(\neg x_1 \vee \neg x_2)$. In fact, only two possible assignments remain. The truth assignment $\{x_1 = 0, x_2 = 1\}$ satisfies the same four clauses as in the previous (non-partial) example, and $\{x_1 = 0, x_2 = 0\}$ satisfies only three clauses. It follows that the optimal solution for the formula is $\{x_1 = 0, x_2 = 1\}$. This solution has cost 2 from the two clauses not satisfied by it.

Weighted Partial MaxSAT A second common extension of SAT augments a CNF formula with weights. Weighted MaxSAT associates a positive weight (or cost) with each $C \in \mathcal{F}$. Formally, it specifies a *cost function* $c : \mathcal{F} \to \mathbb{R}^+$. Both MaxSAT and partial MaxSAT can be viewed as special cases of weighted MaxSAT in terms of optimal solutions. A MaxSAT problem is a weighted MaxSAT problem with $c(C) = 1$ for all $C \in \mathcal{F}$. A partial MaxSAT problem is a weighted MaxSAT problem where $c(C) = 1$ for all $C \in \mathcal{F}_s$ and $c(C) = |\mathcal{F}_s| + 1$ for all $C \in \mathcal{F}_h$, with the additional consideration that no τ for which $\mathit{cost}(\tau) > |\mathcal{F}_s|$ is a valid solution. Alternatively, one could let $c(C) = \infty$ for all $C \in \mathcal{F}_h$. An optimal truth assignment $\tau = \mathit{opt}(\mathcal{F})$ is now one that minimizes the total cost of the unsatisfied clauses, $\mathit{cost}(\tau) = \sum_{C \in \mathcal{F}}\big((1 - \tau(C)) \cdot c(C)\big)$. The limitations of many MaxSAT algorithms mean that clause weights are commonly restricted to integer values, but the methods this work focuses on have no such limitation.

It is common to combine the partial and weighted extensions into weighted partial MaxSAT. This is the most general widely considered variant of the MaxSAT problem, and the variant which we will focus on. Predictably, it combines the distinction between hard and soft clauses of partial MaxSAT with the cost function of weighted MaxSAT. Formally, a weighted partial MaxSAT instance $(\mathcal{F}_h, \mathcal{F}_s, c)$ consists of a set of hard clauses $\mathcal{F}_h$, a set of soft clauses $\mathcal{F}_s$, and a cost function $c : \mathcal{F}_s \to \mathbb{R}^+$. An optimal solution minimizes the cost of unsatisfied soft clauses $\mathit{cost}(\tau) = \sum_{C \in \mathcal{F}_s}\big((1 - \tau(C)) \cdot c(C)\big)$ and satisfies the hard clauses $\mathcal{F}_h$.

Updating our running example formula with weights, and using the notation (C, w) to denote a soft clause C with weight $c(C) = w$, we create a weighted partial MaxSAT instance

$$\begin{aligned} \mathcal{F}_h &= (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2), \\ \mathcal{F}_s &= (x_1 \vee \neg x_2, 3) \wedge (x_1 \vee x_2, 1) \wedge (x_1, 1) \wedge (x_2, 1). \end{aligned} \tag{3}$$

We look again at the viable variable assignments for the partial MaxSAT instance of Equation 2. Given the new weights, the previously optimal model $\{x_1 = 0, x_2 = 1\}$ now has cost 4, as it does not satisfy the weighted soft clauses $(x_1 \vee \neg x_2, 3)$ and $(x_1, 1)$. Our other viable assignment in terms of the hard clauses, $\{x_1 = 0, x_2 = 0\}$, does not satisfy soft clauses $(x_1 \vee x_2, 1)$, $(x_1, 1)$, and $(x_2, 1)$. These unsatisfied clauses have total cost 3. In this case there are no other models which satisfy $\mathcal{F}_h$, so $\{x_1 = 0, x_2 = 0\}$ must be optimal.

2.3.1 Unsatisfiable cores

A final consideration to be addressed before discussing MaxSAT algorithms is the concept of unsatisfiable subsets of clauses. An unsatisfiable subset, or *core*, of a partial MaxSAT instance $(\mathcal{F}_h, \mathcal{F}_s)$ is a set of clauses $\kappa \subseteq \mathcal{F}_s$ such that $\mathcal{F}_h \cup \kappa$ is unsatisfiable. A core κ is a minimal unsatisfiable subset (MUS) if for every proper subset $\kappa' \subset \kappa$, we have that $\kappa' \cup \mathcal{F}_h$ is satisfiable. This general definition also extends to plain MaxSAT and SAT, where $\mathcal{F}_h = \emptyset$. In other words, a core of a plain MaxSAT instance is simply an unsatisfiable subformula. To illustrate the concept, let $\mathcal{F} = (\mathcal{F}_h, \mathcal{F}_s)$ be the partial MaxSAT formula of Equation 2:

$$\begin{aligned}\mathcal{F}_h &= (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2),\\ \mathcal{F}_s &= (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1) \wedge (x_2).\end{aligned}$$

Here $\kappa = \{(x_1 \vee \neg x_2), (x_1 \vee x_2)\}$ is a core, because $\mathcal{F}_h \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is unsatisfiable. Furthermore, κ is minimal because both $\mathcal{F}_h \wedge (x_1 \vee \neg x_2)$ and $\mathcal{F}_h \wedge (x_1 \vee x_2)$ are satisfiable, as they have satisfying assignments $\{x_1 = 0, x_2 = 0\}$ and $\{x_1 = 0, x_2 = 1\}$. On the other hand, the set of soft clauses $\{(x_1 \vee x_2), (x_1), (x_2)\}$ is also a core, but it is not minimal because its subset $\{(x_1), (x_2)\}$ is a core.

Finally, we show that there exist partial MaxSAT instances with exponentially many minimal cores.

Theorem 3. *The number of cores in a partial MaxSAT instance* $(\mathcal{F}_h, \mathcal{F}_s)$ *can be exponential in* $m = |\mathcal{F}_s|$.

Proof. Let $\mathcal{F}_s = \bigwedge_{i=1}^{m} C_i$ and let $\mathcal{F}_h$ be a CNF encoding of $\sum_{C_i \in \mathcal{F}_s} C_i < \frac{m}{2}$. No assignment which satisfies $\frac{m}{2}$ soft clauses satisfies $\mathcal{F}_h$, and as such any set of $\frac{m}{2}$ soft clauses is a (minimal) core. There are

$$\binom{m}{m/2} = \frac{m!}{\frac{m}{2}!\frac{m}{2}!} = \frac{\prod_{i=m/2+1}^{m} i}{\frac{m}{2}!} = \prod_{i=1}^{m/2} \frac{\frac{m}{2} + i}{i} > 2^{m/2}$$

such cores. □

2.3.2 MaxSAT solvers

This section gives a short overview of techniques for solving MaxSAT. Modern MaxSAT algorithms can for the most part be roughly categorized into four categories. So-called SAT based algorithms [3], which iteratively use a SAT solver, can be split into *satisfiability-based* and *unsatisfiability-based* approaches. These iterative SAT-based algorithms often rely on CNF encodings of cardinality constraints, that is, constraints of the form $\sum_{i=0}^{n} x_i \leq k$. This type of constraint sees use in many applications, which has led to a considerable amount of work [10, 5, 30, 45, 80, 104] towards finding efficient CNF encodings for them. A third category of *branch-and-bound algorithms* [72] have also been applied to MaxSAT successfully in some domains. A fourth approach is *reformulating* a MaxSAT instance or some subproblem into another problem domain such as IP [101]. As with SAT solvers, international competitive events have helped push the development of algorithms for MaxSAT. Since 2006, the yearly MaxSAT Evaluations[1] have assessed the state-of-the-art in MaxSAT solvers.

Unsatisfiability-based algorithms A common type of algorithm for MaxSAT begins with an unsatisfiable working SAT formula $\mathcal{F}$ and gradually relaxes it until the working formula becomes satisfiable. This is typically done by identifying unsatisfiable cores in $\mathcal{F}$ and modifying the CNF encoding to remove the individual core. The sequence of unsatisfiable SAT problems corresponds to an increasing lower bound for the optimal MaxSAT solution. As an example, we examine the archetypal core-based procedure of Fu and Malik [48], outlined in Algorithm 2.

Algorithm 2 The Fu–Malik core-based partial MaxSAT algorithm.

Require: $\mathcal{F}_h$ is satisfiable.

1: **function** FuMalik($\mathcal{F}_h, \mathcal{F}_s$)
2: **while** True **do**
3: $(sat, \kappa, \tau) \leftarrow$ SolveSAT($\mathcal{F}_h \cup \mathcal{F}_s$)
4: **if** *sat* **then break**
5: $A \leftarrow \emptyset$
6: **for** $C \in \kappa$ **do**
7: Allocate a new relaxation variable a.
8: $\mathcal{F}_s \leftarrow (\mathcal{F}_s \setminus \{C\}) \cup \{(C \vee a)\}$
9: $A \leftarrow A \cup a$
10: $\mathcal{F}_h \leftarrow \mathcal{F}_h \cup CNF(\sum_{a \in A} a = 1)$
11: **return** τ

Algorithm 2 uses a SAT solver to identify unsatisfiable cores. Each core κ

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

is relaxed by adding new relaxation variables a to each of its clauses (Line 6) and adding a hard constraint (Line 10) requiring that only one of these relaxation variables be true. This process is repeated until no more cores are found, i.e., the relaxed formula becomes satisfiable. A drawback of this method is that a new variable is added to a clause for *every* core it is found in. Together with the added hard constraints, this means that each subsequent SAT solver call is on a more complex formula.

The Fu–Malik algorithm was extended and improved on in [83, 4], and implemented in, e.g., the solvers MSUnCore and WPM1. A more recent core-based solver, eva500a [92] solved the most industrial weighted partial MaxSAT instances in the 2014 MaxSAT Evaluation.

Satisfiability-based algorithms Another category of algorithms begins with a satisfiable SAT problem and iteratively constrains it until the formula becomes unsatisfiable. The sequence of satisfiable SAT problems corresponds to a decreasing upper bound for the MaxSAT problem. Algorithm 3 outlines a simple satisfiability-based MaxSAT algorithm, which uses a linear search to find an optimal solution. Conceptually similar algorithms are found in, e.g., the linear search algorithm of Open-WBO [87] and the QMaxSAT [70] solver.

Algorithm 3 A satisfiability-based linear search algorithm for MaxSAT.

Require: $\mathcal{F}$ is unsatisfiable.

```
1:  function LinearMaxSAT(F)
2:      F' ← {(C_i ∨ a_i) : C_i ∈ F}
3:      Card ← ∅
4:      τ' ← ∅
5:      while True do
6:          (sat, τ) ← SolveSAT(F' ∪ Card)
7:          if not sat then break
8:          k ← "number of a_i variables true in τ"
9:          Card ← CNF(Σ_{i=0}^{n} a_i < k)
10:         τ' ← τ
11:     return τ'
```

Algorithm 3 uses satisfiable SAT problems to perform a linear search for the optimal MaxSAT solution. For simplicity it is presented here for plain MaxSAT, but is easily extended to partial MaxSAT by relaxing only the soft clauses the instance. The algorithm begins by relaxing every clause of a MaxSAT problem with a unique relaxation variable a_i (Line 2). The resulting formula can clearly be satisfied by any model in which all of the newly added relaxation variables are true. The algorithm maintains a cardinality constraint $Card$ that is updated with every found model (Line 9). The purpose of the constraint is to check whether there exists a model which

has fewer true relaxation variables than the previous model. Once this cardinality constraint makes the SAT formula unsatisfiable, we know that the previous model is an optimal one.

Branch-and-bound Solvers based on branch-and-bound have traditionally been most effective on unweighted and random MaxSAT instances, but recent MaxSAT evaluations have shown them to be comparatively ineffective on real-world weighted partial instances. Some solvers of note include ahmaxsat [1], which won several categories in the 2014 MaxSAT Evaluation and, MaxSatz [75], which was the best solver for unweighted MaxSAT in the 2007 evaluation. Another interesting branch-and-bound approach was used in clone [97], which derived bounds by compiling relaxed formulas to the tractable language of d-DNNF [34].

Reformulation and hybrid approaches Yet another approach is to reformulate a MaxSAT instance as an instance of another optimization problem. Section 2.4 provides an example of such an approach with an integer programming reformulation of MaxSAT.

The focus of this work is on MaxHS, a hybrid approach. Like most unsatisfiability-based algorithms, MaxHS identifies unsatisfiable cores of the MaxSAT instance. However, it differs from the algorithms already discussed in this section in several ways. The algorithm adds no clauses to the formula and does not require any additional cardinality constraints to be encoded in CNF. It uses both a SAT solver and an integer programming solver, identifying the unsatisfiable cores with the SAT solver and essentially incrementally reformulating the optimization aspect of MaxSAT as an IP problem. Section 3 examines the algorithm in more detail.

2.4 Integer programming

Integer programming (IP) problems are linear programming (LP) problems with the additional constraint that variables must take integral values. Solving an IP problem involves finding an integral solution to a system of linear equations, which maximizes or minimizes some objective function. Unlike LP, for which polynomial time algorithms exist [68, 64, 41], IP is known to be NP-complete [101]. In this work our interest in IP is primarily limited to using IP solvers as black boxes, not the specific methods by which IP problems are solved. Section 3 focuses specifically on the MaxHS algorithm, which uses an IP solver internally for solving specific sub-problems. Section 6 looks at modifying an IP-based method for optimal Bayesian network learning.

In integer programming, we seek to maximize or minimize a linear objec-

tive function f of the form

$$f(x_1, \ldots, x_n) = w_1 x_1 + \cdots + w_n x_n,$$

where $x_1, \ldots, x_n$ are the problem variables and $w_1, \ldots, w_n$ are some fixed weights. This optimization is subject to a set of linear constraints of the form

$$a_1 x_1 + \cdots + a_n x_n \leq k \quad \text{or} \quad a_1 x_1 + \cdots + a_n x_n \geq k$$

with fixed coefficients a_i and k. An IP problem will additionally constrain (some of) the variables $x_1, \ldots, x_n$ to integers, or some subset of integers. In this work, binary variables will often be used, giving us the integrality constraint

$$x_i \in \{0, 1\} \text{ for all } x_i.$$

As an example, consider the following reformulation of weighted partial MaxSAT as integer programming [74]. Let $\mathcal{F} = (\mathcal{F}_h, \mathcal{F}_s, c)$ be a MaxSAT instance with variables $x_1, \ldots, x_n$ and $|\mathcal{F}_s| = m$. Now introduce new binary variables $y_1, \ldots, y_m$ corresponding to the soft clauses $C_i \in \mathcal{F}_s$ and let $c(y_i) = c(C_i)$. Now the IP can be expressed as

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{m} y_i \cdot c(y_i), & \\
\text{subject to} \quad & \sum_{x_i \in C_j} x_i - \sum_{\neg x_i \in C_j} x_i + y_j \geq 1 & \forall C_j \in \mathcal{F}_s, \\
& \sum_{x_i \in C_j} x_i - \sum_{\neg x_i \in C_j} x_i \geq 1 & \forall C_j \in \mathcal{F}_h, \\
& x_i, y_j \in \{0, 1\} & \forall x_i, \forall y_j.
\end{aligned}$$

In other words, we assert that each hard clause must have at least one satisfied literal. Each soft clause must either have a satisfied literal or its corresponding y_i variable must be 1. These y_i variables are used to construct the objective function under minimization.

3 The MaxHS algorithm

MaxHS is a recent core-based algorithm for solving MaxSAT. The concept was initially introduced by Jessica Davies and Fahiem Bacchus in [37] and further refined in [38, 39, 36]. A solver based on the MaxHS algorithm was able to solve the most weighted partial MaxSAT instances in the crafted category of the 2013 MaxSAT Evaluation[2]. The algorithm is differentiated from other successful MaxSAT algorithms by its hybrid approach. It essentially separates the satisfiability and optimization parts of MaxSAT so that

[2] `http://www.maxsat.udl.cat/13/`

a suitable method can be used to solve each part. A SAT solver is used to identify unsatisfiable cores, and an IP solver is used to compute minimum cost hitting sets for these cores. This can be thought of as an instantiation of a general implicit hitting set approach to optimization [27, 88] to MaxSAT. Purely SAT-based MaxSAT solvers solve a sequence of problems which tend to get harder with each iteration as more constraints are added to the CNF formula. The separation between SAT and IP in MaxHS allows us to avoid introducing additional constraints to the CNF formula. In fact, the SAT problems solved by MaxHS tend to get easier with each iteration of the algorithm.

3.1 Hitting sets

The MaxHS algorithm relies on an integer programming (IP) solver to find minimum-cost hitting sets for sets of unsatisfiable cores of a weighted partial MaxSAT instance. Given a set $S = \{s_1, \ldots, s_n\}$ of sets, a *hitting set* (HS) of S is a set H such that $H \cap s_i \neq \emptyset$ for all $s_i \in S$. In other words, some element of every s_i must be "hit" by an element of H. Given a cost function $c : S \to \mathbb{R}^+$, a *minimum-cost hitting set* (MCHS) for S is a hitting set that minimizes $\sum_{s_i \in H} c(s_i)$ over all hitting sets of S. The hitting set problem is equivalent to the set cover problem, which is NP-complete in its decision problem form [65]. A well-known IP formulation of MCHS, presented in the context of MaxHS, follows. Let $(\mathcal{F}_h, \mathcal{F}_s, c)$ be a weighted partial MaxSAT instance. Each core κ is a subset of $\mathcal{F}_s$ and $\mathcal{K}$ is a set of cores. Now we can express the IP as

$$\begin{aligned} \text{minimize} \quad & \sum_{C \in \mathcal{F}_s} c(C) \cdot x_C, & & \\ \text{subject to} \quad & \sum_{C \in \kappa} x_C \geq 1 & & \forall \kappa \in \mathcal{K}, \\ & x_C \in \{0, 1\} & & \forall C \in \mathcal{F}_s. \end{aligned} \tag{4}$$

We introduce a binary variable x_C for each clause C in $\mathcal{F}_s$. A solution with $x_C = 1$ signifies that C is in the corresponding hitting set H. Conversely, $C \notin H$ if $x_C = 0$. Equation 4 introduces for each core $\kappa \in \mathcal{K}$ a constraint requiring that $x_C = 1$ for at least one $C \in \kappa$. The objective is then to minimize the sum of the costs of clauses C for which $x_C = 1$.

Minimum-cost hitting set approximations are also of interest for MaxHS, as detailed in Section 3.4. To that end, Algorithm 4 adapts the greedy set cover algorithm of [61] to the weighted hitting set problem. Algorithm 4 begins by initializing a working hitting set H and a set of unhit cores U (Lines 2 and 3). While unhit cores remain, the algorithm gathers (Line 5) the set of elements in unhit cores and greedily chooses (Line 6) the element e that maximizes the ratio of cost $c(e)$ to the number of previously unhit sets hit by e. The subroutine $\textsc{Count}(U, e)$ returns the number of sets in

Algorithm 4 A greedy hitting set algorithm.

```
1: function GreedyHS(𝒦, c)
2:     H ← ∅
3:     U ← 𝒦
4:     while U ≠ ∅ do
5:         E ← ⋃_{κ∈U} κ
6:         e ← arg max_{e∈E}(c(e)/Count(U, e))
7:         H ← H ∪ {e}
8:         U ← U \ {κ ∈ U : e ∈ κ}
9:     return H
```

U which are hit by e. The chosen element e is then added (Line 7) to the hitting set and every core κ hit by e is removed (Line 8) from U.

3.2 Solving MaxSAT with cores and hitting sets

The MaxHS algorithm is based on the principle that a minimum-cost hitting set for the cores of a MaxSAT instance corresponds to an optimal MaxSAT solution. The intuition behind this is simple: if we know every possible unsatisfiable subset of clauses and a MCHS over these subsets, there must exist a MaxSAT solution which satisfies every clause *not* in the hitting set. Furthermore, because the hitting set is of minimum weight, this MaxSAT solution has maximum weight among all solutions. This type of algorithm can be traced at least as far back as [99], in which an algorithm which finds a hitting set over so-called "conflict sets" for diagnostic reasoning problems is suggested.

A key observation (Theorem 4) here is that it is often not necessary to find *every* core to find an optimal MaxSAT solution. If the cores of a MaxSAT instance have significant overlap, it is likely that a clause from one core will hit multiple cores even if they have not been identified. For a MaxSAT instance $(\mathcal{F}_h, \mathcal{F}_s)$ we need only to find a MCHS H over some set cores $\mathcal{K}$ such that $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$ is satisfiable. That is, it is sufficient to find a MCHS over a set of cores such that removing its clauses from the MaxSAT instance makes it satisfiable.

Theorem 4. *[37] Let $\mathcal{F} = (\mathcal{F}_h, \mathcal{F}_s, c)$ be a weighted partial MaxSAT instance. Assume that $\mathcal{K}$ is a subset of the cores of $\mathcal{F}$, H is a MCHS of $\mathcal{K}$, and τ is a satisfying truth assignment for $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$. Let $\hat{\tau} \in opt(\mathcal{F})$, then*

$$cost(\hat{\tau}) = cost(\tau) = cost(H).$$

Proof. Since $\hat{\tau}$ has minimal cost over all assignments, $cost(\hat{\tau}) \leq cost(\tau)$ must hold. The truth assignment of τ satisfies every clause of $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$, so its cost can be no greater than the cost of the excluded clauses of H,

and thus $cost(\tau) \leq cost(H)$ must hold. Finally, H hits the cores in $\mathcal{K}$ with minimum cost, and $\hat{\tau}$ must leave unsatisfied a clause from *every* core of $\mathcal{F}$ with minimum cost. It follows that $cost(\hat{\tau}) \geq cost(H)$. □

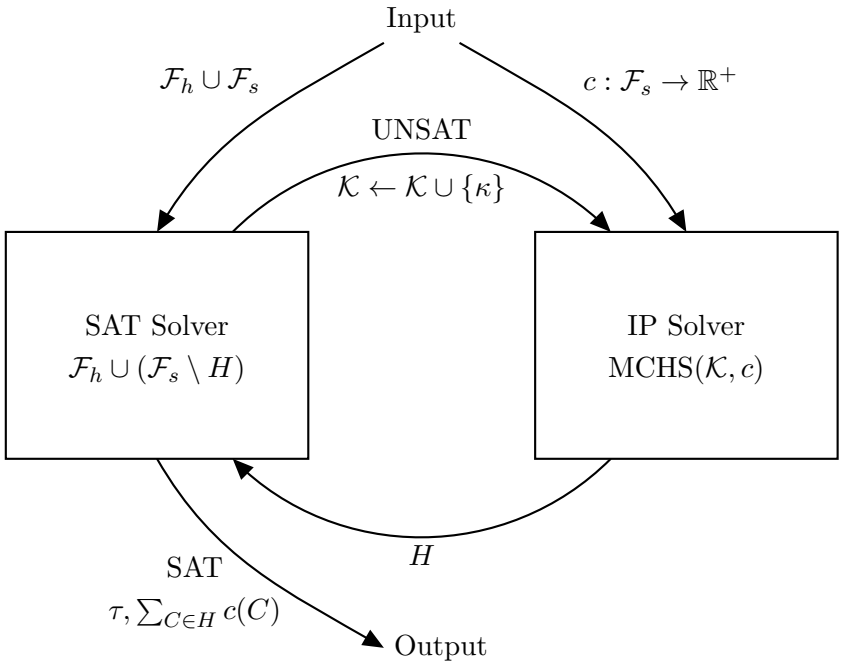


Figure 1: MaxHS information flow.

This observation leads naturally to the MaxHS algorithm. Figure 1 visualizes this concept and the flow of information between the SAT and IP solvers. With a SAT solver which allows us to extract a core of an unsatisfiable CNF formula, we can extract a new core of $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$ until the hitting set H for the resulting set of cores corresponds to an optimal solution. After this, the found satisfying variable assignment will be optimal. The minimum-cost hitting set problem can be solved with an IP solver with the formulation of Algorithm 4. This IP solver needs only the cores (which can be incrementally added to the problem as they are found) and the MaxSAT cost function; it does not require knowledge of the CNF formula. Likewise, the SAT solver can operate independently of the cost function. The visualization of Figure 1 is presented more concretely as Algorithm 5.

Algorithm 5 begins by initializing an empty set of cores and an empty hitting set. An initial SAT solver invocation (Line 4) serves to check the trivial case in which $\mathcal{F}_h \cup \mathcal{F}_s$ is satisfiable. If it is not satisfiable, a core κ is found and the main loop of the algorithm follows. The core κ is added to $\mathcal{K}$ (Line 6) and a minimum-cost hitting set H is computed for $\mathcal{K}$. The SAT solver is then invoked (Line 8) to determine the satisfiability of $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$.

Algorithm 5 A core-based algorithm for MaxSAT using hitting sets [37].

Require: $\mathcal{F}_h$ is satisfiable.

1: **function** MaxHS($\mathcal{F}_h, \mathcal{F}_s, c$)
2: $\quad\mathcal{K} \leftarrow \emptyset$
3: $\quad H \leftarrow \emptyset$
4: $\quad(sat, \kappa, \tau) \leftarrow$ SolveSAT($\mathcal{F}_h \cup \mathcal{F}_s$)
5: $\quad$**while not** sat **do**
6: $\quad\quad\mathcal{K} \leftarrow \mathcal{K} \cup \{\kappa\}$
7: $\quad\quad H \leftarrow$ SolveMCHS($\mathcal{K}, c$)
8: $\quad\quad(sat, \kappa, \tau) \leftarrow$ SolveSAT($\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$)
9: $\quad$**return** τ

Lines 6–8 are repeated until $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$ is satisfiable, at which point τ is an optimal solution by Theorem 4.

It is interesting to note that although the worst-case behavior of MaxHS is in a sense sub-optimal, it has proven to be effective in practice. MaxHS could require an exponential number of SAT and IP solver invocations. This is because cores found in MaxHS are not guaranteed to raise the lower bound, so it is possible that all cores must be found. Theorem 3 states that there can be an exponential number of cores in a MaxSAT instance. Davies and Bacchus show in [39] that there exist instances for which the MaxHS algorithm must also find an exponential number of cores. By contrast, some purely SAT-based approaches require a linear number of SAT solver invocations. However, these approaches must deal with a formula which grows with each iteration. Theorem 5 shows that the MaxHS algorithm will find an optimal solution and terminates.

Theorem 5. *[37] Algorithm 5 finds opt($\mathcal{F}$) and terminates.*

Proof. That the algorithm finds *opt*($\mathcal{F}$) follows from Theorem 4. We note that $\mathcal{F}$ is finite, and as such has a finite number of cores. Each H removes every core of $\mathcal{K}$ from $\mathcal{F}$, so at every iteration κ will be distinct from (and not a superset of) each core already in $\mathcal{K}$. It follows that given sufficient iterations, Algorithm 5 will find every core of $\mathcal{F}$, and the hitting set H will hit each of them. At this point $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$ must be satisfiable, and the algorithm will terminate. □

For a short example execution of MaxHS, we use the following weighted partial MaxSAT instance.

$$\mathcal{F}_h = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2),$$
$$\mathcal{F}_s = (x_1 \vee \neg x_2, 2) \wedge (\neg x_1 \vee \neg x_2, 3) \wedge (x_1, 4) \wedge (x_2, 4) \wedge (\neg x_1, 2).$$

Table 1 shows a possible execution of the MaxHS algorithm for the instance $(\mathcal{F}_h, \mathcal{F}_s)$. It shows a core found at each iteration, the minimum-cost hitting set

computed for all cores found up to that iteration, and the cost of the hitting set. Recall that the satisfiability of a partial MaxSAT core is considered in conjunction with the set of hard clauses. At the fifth iteration, no core is found and an optimal solution has been found. The example of Table 1 illustrates the fact that the MCHS does not necessarily grow in size with the number of cores found. It also provides an example of implicitly hit cores. The cores $\{(\neg x_1), (x_1 \vee \neg x_2)\}$ and $\{(x_1), (\neg x_1 \vee \neg x_2)\}$ are hit by the final hitting set despite never having been explicitly identified. We also note that the cores need not be minimal as, e.g., the set of clauses $\{(x_1), (\neg x_1 \vee \neg x_2)\} \subset \{(x_1), (x_2), (\neg x_1 \vee \neg x_2)\}$ is a core.

Iteration	Core	MCHS	MCHS cost
1	$\{(x_1 \vee \neg x_2), (\neg x_1 \vee \neg x_2)\}$	$\{(x_1 \vee \neg x_2)\}$	2
2	$\{(x_1), (x_2)(\neg x_1 \vee \neg x_2)\}$	$\{(\neg x_1 \vee \neg x_2)\}$	3
3	$\{(\neg x_1), (x_2), (x_1 \vee \neg x_2)\}$	$\{(\neg x_1 \vee \neg x_2), (x_1 \vee \neg x_2)\}$	5
4	$\{(x_1), (\neg x_1)\}$	$\{(x_1), (x_1 \vee \neg x_2)\}$	6
5	–	$\{(x_1), (x_1 \vee \neg x_2)\}$	6

Table 1: MaxHS algorithm execution.

3.3 Assumption variables in MaxHS

When possible, algorithms and techniques will be presented on the abstraction level of Algorithm 5. However, in the context of this work it is necessary to consider one specific detail of the implementation: the use of assumptions and assumptions variables in solving the sequence of SAT problems [43]. Assumptions can be typically treated as a transparent aspect of the algorithm implementation, but Sections 3.5.2 and 5.4 discuss how they can be exploited in the solving process.

The MaxHS procedure is dependent on the ability to efficiently remove and reinsert soft clauses into the working formula. With each hitting set, Line 8 of Algorithm 5 must temporarily remove some clauses from the SAT solver. Explicit removal of a clause C is not practical in CDCL SAT solvers due to the difficulty of determining which learned clauses depend on C. Many solvers [44, 22, 50] offer an alternative in the form of assumptions.

A set of assumptions is a partial assignment to the variables of a formula. This partial assignment is treated as permanent top-level decisions in the CDCL algorithm for an invocation of the solver. Given a formula $\mathcal{F} = \mathcal{F}_h \cup \mathcal{F}_s$, assumptions can be used to temporarily remove (or disable) soft clauses in successive SAT calls. It is first necessary to introduce an *assumption variable*[3], a previously unused binary variable a_i, for each soft clause in $\mathcal{F}_s$. Then a new formula $\mathcal{F}' = \mathcal{F}_h \cup \mathcal{F}'_s$ can be created, where $\mathcal{F}'_s = \bigcup_{C_i \in \mathcal{F}_s} \{(C_i \vee a_i)\}$.

[3]Assumption variables are also known as relaxation variables, blocking variables, or labels

Algorithm 6 CDCL with assumptions

```
1: function AssumptionCDCL(F, A)
2:     D ← A
3:     τ ← ∅
4:     while True do
5:         (c, τ) ← Propagate(F, D, τ)
6:         if c contains a conflict then
7:             if |D| = |A| then
8:                 return AnalyzeFinalConflict(c)
9:             else                                  ▷ learn a conflict clause
10:                C ← AnalyzeConflict(c)
11:                F ← F ∪ {C}
12:                (D, τ) ← UndoDecisions(D, τ, C)        ▷ backjump
13:        else if τ(F) = 1 then
14:            return Satisfiable, τ
15:        else                                      ▷ make a new decision
16:            Choose a variable x unassigned by τ
17:            Make a decision for x: d ← {x = 0} or d ← {x = 1}
18:            D ← D ∪ {d}
```

When used to solve $\mathcal{F}'$, the SAT solver can be given a set of assumptions over the new variables a_i. Assuming $a_i = 0$ "enables" the corresponding original clause C_i as the only way to satisfy $(C_i \vee a_i)$ is to satisfy C_i. Assuming $a_i = 1$ "disables" C_i by satisfying $(C_i \vee a_i)$, which leaves the solver free to ignore C_i. These assumptions can be combined to exclude an arbitrary set of soft clauses, allowing us to solve a subformula of $\mathcal{F}$ without permanently removing clauses.

Recall the CDCL procedure of Algorithm 1. Algorithm 6 extends it to determine the satisfiability of a formula $\mathcal{F}$ under a set of assumptions A. Some key changes to plain CDCL are necessary to solve under assumptions and to evaluate unsatisfiability in terms of assumptions. The assumptions in A are used as a base-level variable assignment in the CDCL search. These are permanent for the duration of the search, and UndoDecisions is not permitted to remove them. The CDCL termination condition for unsatisfiability is also modified. Rather than reporting unsatisfiability when a conflict is derived under no assignments, this is done when only the initial assignments of the assumptions are left. Finally, AnalyzeFinalConflict, a specialization of AnalyzeConflict, will produce a conflict clause containing only literals in A, constituting an unsat core of $\mathcal{F}$ under A. This functionality is available for example in the commonly used MiniSat [44] SAT-solver.

Algorithm 7 re-frames MaxHS in terms of assumptions. When considering assumption variables, we temporarily abandon the clause-centric view

Algorithm 7 MaxHS with assumptions.

Require: $\mathcal{F}_h$ is satisfiable.

1: **function** MaxHS($\mathcal{F}_h, \mathcal{F}_s, c$)
2: $\quad\mathcal{F}' \leftarrow \mathcal{F}_h \cup \{(C_i \vee a_i) : C_i \in \mathcal{F}_s\}$ ▷ Relax $\mathcal{F}$ with assumptions.
3: $\quad\mathcal{K} \leftarrow \emptyset$
4: $\quad H \leftarrow \emptyset$
5: $\quad(sat, \kappa, \tau) \leftarrow$ SolveWithAssumptions($\mathcal{F}', \bigcup_{i=0}^{m}\{a_i = 0\}$)
6: $\quad$**while not** sat **do**
7: $\quad\quad\mathcal{K} \leftarrow \mathcal{K} \cup \{\kappa\}$
8: $\quad\quad H \leftarrow$ SolveMCHS($\mathcal{K}, c$)
9: $\quad\quad A \leftarrow \{a_i = 1 : a_i \in H\} \cup \{a_i = 0 : a_i \notin H\}$)
10: $\quad\quad(sat, \kappa, \tau) \leftarrow$ SolveWithAssumptions($\mathcal{F}', A$)
11: $\quad$**return** τ

of Algorithm 5. Specifically the cores, hitting sets, and cost function in Algorithm 7 are expressed in terms of assumption variables instead of their corresponding clauses.

3.4 Using non-optimal hitting sets

Every iteration of Algorithm 5 increases the size of the minimum-cost hitting set problem that is given to the IP solver. The IP solver must optimally solve an instance of an NP-hard problem, so it seems natural to try to avoid these solver calls whenever possible. The technique of deriving cores from non-optimal hitting sets within MaxHS was introduced in [39] for this purpose.

To preserve correctness and the optimality of the result, only the last minimum-cost hitting set computed for MaxHS—which yields a satisfiable formula—must be optimal. The optimality of a hitting set H has no effect on the validity of a core derived from $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$, only on the existence of cores. In other words, any core of $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$ is also a core of $\mathcal{F}_h \cup \mathcal{F}_s$. This means that approximation strategies can be used to compute some hitting sets in polynomial time for the purpose of extracting cores more efficiently.

Algorithm 8 Finding cores via non-optimal hitting sets.

1: **function** NonOpt($\mathcal{F}_h, \mathcal{F}_s, c, \mathcal{K}, H$)
2: $\quad$**repeat**
3: $\quad\quad H \leftarrow$ NonOptimalHS($\mathcal{K}, c$)
4: $\quad\quad(sat, \kappa, \tau) \leftarrow$ SolveSAT($\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$)
5: $\quad\quad$**if not** sat **then** $\mathcal{K} \leftarrow \mathcal{K} \cup \{\kappa\}$ **end if**
6: $\quad$**until** sat
7: $\quad$**return** $\mathcal{K}$

Different heuristics can be applied here to try to find a "good" hitting set H in the sense that $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$ is unsatisfiable and thus yields a core. Optimal hitting sets are still necessary for MaxHS, but only to verify (by Theorem 4) that a MaxSAT solution is optimal or to find cores when a non-optimal hitting set does not. Algorithm 8 shows a procedure for utilizing arbitrary non-optimal hitting sets to find additional cores.

Various techniques exist for computing these non-optimal hitting sets. The greedy hitting set procedure of Algorithm 4 is one such technique. A non-optimal hitting set can also be constructed incrementally from an existing hitting set, for example by inserting into it the clause of a new core κ which is most common among the cores of $\mathcal{K}$. These techniques are combined in [37] into a two-phase non-optimal hitting set strategy as shown in Algorithm 9.

Algorithm 9 Using non-optimal minimum-cost hitting sets in two phases.

```
1:  function NonOpt2(F_h, F_s, c, K, κ, H)
2:      repeat
3:          repeat
4:              H ← IncrementalHS(κ, H, c)
5:              (sat, κ, τ) ← SolveSAT(F_h ∪ (F_s \ H))
6:              if not sat then K ← K ∪ {κ}
7:          until sat
8:          H ← GreedyHS(K, c)
9:          (sat, κ, τ) ← SolveSAT(F_h ∪ (F_s \ H))
10:         if not sat then K ← K ∪ {κ}
11:     until sat
12:     return K
```

Here a distinction is made between IncrementalHS which computes a new hitting set by simply adding an element from a new core κ to an existing hitting set, and GreedyHS (recall Algorithm 4), which computes a new hitting set for a set of cores $\mathcal{K}$. It is also possible to construct a non-optimal hitting set by adding more than one clause from a newly found core to an existing hitting set. Davies finds in [36] that adding 10% of the most common clauses (among other found cores) in a new core to the hitting set performs well as a heuristic. We empirically investigate the behavior of this type of non-optimal hitting set strategies in Section 4.2.

3.5 Presolving

Davies and Bacchus established in [38] the benefits of finding an initial set of constraints for the IP solver as a presolving step before Algorithm 5. As a result of the hybrid approach of the solver, neither the SAT nor the IP solver alone has sufficient information to solve a MaxSAT instance. While the SAT

solver works on the entire CNF formula from the start, in Algorithm 5 the IP solver has no knowledge of the problem prior to the first iteration. The intuition behind finding these initial cores and constraints is to reduce the number of potentially costly IP solver calls by allowing the solver to make more informed decisions. In this section we examine two such methods: an initial phase for finding disjoint cores and assumption variable equivalences.

3.5.1 Disjoint phase

Algorithm 10 Finding a maximal disjoint set of cores in a partial MaxSAT instance $(\mathcal{F}_h, \mathcal{F}_s)$.

```
1: function DisjointCores(F_h, F_s)
2:     F' ← F_h ∪ F_s
3:     K ← ∅
4:     repeat
5:         (sat, κ, τ) ← SolveSAT(F')
6:         if not sat then
7:             K ← K ∪ {κ}
8:             F' ← F' \ κ
9:     until sat
10:    return K
```

A potentially large set of cores can be found without needing to calculate hitting sets. One way to accomplish this is outlined in Algorithm 10, which finds a disjoint set of cores [38]. After finding a core κ, we can remove every clause in κ from the working formula $\mathcal{F}'$, and use the SAT solver to find a new core in the reduced formula. If another core is found, it will have no overlap with the previous core, and the process can be repeated. Once no more cores are found in the remaining formula, a maximal set of disjoint cores has been found. Beyond saving IP solver calls, this method is comparatively inexpensive as it reduces the size of the formula that must be refuted after every found core. The cores found are also effective at increasing a lower bound for the optimal solution because they are disjoint. The minimum-cost hitting set must hit a unique clause for each core, so each disjoint core will increase the know lower bound for the instance. It is also more likely that unfound cores will be implicitly hit because the size of a minimum-cost hitting set is equal to the number of sets when the sets are disjoint. This initial phase is also computationally efficient because finding a minimum-cost hitting set for a disjoint set of cores is trivial. For example, the weighted partial MaxSAT instance of Section 3.2,

$$\mathcal{F}_h = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2),$$
$$\mathcal{F}_s = (x_1 \vee \neg x_2, 2) \wedge (\neg x_1 \vee \neg x_2, 3) \wedge (x_1, 4) \wedge (x_2, 4) \wedge (\neg x_1, 2),$$

has a maximal disjoint set of cores $\{\{(x_1, \neg x_2), (\neg x_1, \neg x_2)\}, \{(x_1), (\neg x_1)\}\}$. The minimum cost hitting set for these cores, $\{(\neg x_1), (x_1, \neg x_2)\}$, gives a lower bound of 4 for the solution. Other cores for the instance, such as $\{(x_1, \neg x_2), (\neg x_1)\}$ or $\{(\neg x_1, \neg x_2), (x_1)\}$, all overlap with a core in this disjoint set.

3.5.2 Assumption variable equivalences

Relaxing the original MaxSAT instance with assumption variables as described in Section 3.3 adds new structure to the formula. Various ways of extracting constraints usable in IP solving from this relaxed formula were explored in [38]. We examine the most successful of these methods, assumption variable equivalences. Consider for example a MaxSAT instance containing the clauses

$$\begin{aligned} \mathcal{F}_h &= (x_1 \vee \neg x_2 \vee x_3) \wedge \cdots \\ \mathcal{F}_s &= (x_1) \wedge (x_2) \wedge (\neg x_3) \wedge \cdots \end{aligned} \tag{5}$$

We augment the soft clauses with fresh assumption variables to get clauses

$$\mathcal{F}_{S'} = (x_1 \vee a_1) \wedge (x_2 \vee a_2) \wedge (\neg x_3 \vee a_3) \wedge \cdots \tag{6}$$

In the context of an optimal solution produced by Algorithm 5, it also implicitly holds that $(\neg x_1 \vee \neg a_1)$, $(\neg x_2 \vee \neg a_2)$, and $(x_3 \vee \neg a_3)$. To see that this is the case, recall that each assumption variable a_i is found in exactly one clause of $\mathcal{F}_h \cup \mathcal{F}_{S'}$. A value is assumed for every a_i for any SAT solver invocation. If we assume $a_i = 1$, it is because a minimum-cost hitting set H contains a_i. A minimum-cost hitting set is also minimal in size, so for every $a_i \in H$ it must hold that the soft clause C_i corresponding to a_i cannot be satisfied given the assumptions $\{a_j = 0 : a_j \notin H\}$. In other words $\neg C_i \rightarrow a_i$, which gives us $(\neg x_1 \vee \neg a_1)$, $(\neg x_2 \vee \neg a_2)$, and $(x_3 \vee \neg a_3)$.

Together with these implicit constraints, the clauses of Equation 6 now give us equivalences $(x_1 \leftrightarrow \neg a_1)$, $(x_2 \leftrightarrow \neg a_2)$, and $(x_3 \leftrightarrow a_3)$. It follows that the hard clause $(x_1 \vee \neg x_2 \vee x_3)$ is equivalent to $(\neg a_1 \vee a_2 \vee \neg a_3)$. This new clause contains only assumption variable literals, and as such can be used by the IP solver to constrain the search for minimum-cost hitting sets.

3.6 Core minimization

The unsatisfiable cores produced by a SAT solver are not necessarily minimal. In other words, for a core κ, it could be the case that some $\kappa' \subset \kappa$ is also a core. Recall that a core κ for which no such $\kappa' \subset \kappa$ exists is called a minimal unsatisfiable subset or MUS of the instance. In this section we review ways of minimizing or reducing the size of found cores in MaxHS. Any core which is not a MUS can potentially increase the number of minimum-cost hitting sets the algorithm needs to compute. If κ is included in the hitting set problem

but κ' is not, then the IP solver can find a solution H for which $H \cap \kappa' = \emptyset$. This ensures that $\mathcal{F} \setminus H$ is not an optimal solution for the MaxSAT instance, and hence further iterations of the algorithm are required.

To further motivate this search for smaller cores, consider as an example the weighted MaxSAT instance

$$(\neg x_1 \vee x_2, 7) \wedge (\neg x_1 \vee \neg x_2, 8) \wedge (x_1 \vee \neg x_2, 7) \wedge (x_1 \vee x_2, 3) \wedge (x_1, 3) \wedge (x_2, 3). \quad (7)$$

We refer to the clauses of Equation 7 as $c_1, \ldots, c_6$ for convenience. As a worst case, consider how the search of Algorithm 5 could proceed if the SAT solver finds *maximal* cores in Table 2. The table shows the sequence of cores and minimum-cost hitting sets found by iterations of the algorithm. A dash $(-)$ is used to denote that no core was found and $\mathcal{F} \setminus H$ is satisfiable.

Iteration	Core	MCHS	MCHS cost
1	$\{c_1, c_2, c_3, c_4, c_5, c_6\}$	$\{c_6\}$	3
2	$\{c_1, c_2, c_3, c_4, c_5\}$	$\{c_5\}$	3
3	$\{c_1, c_2, c_3, c_4, c_6\}$	$\{c_4\}$	3
4	$\{c_1, c_2, c_3, c_5, c_6\}$	$\{c_5, c_6\}$	6
5	$\{c_1, c_2, c_3, c_4\}$	$\{c_4, c_6\}$	6
6	$\{c_1, c_2, c_3, c_5\}$	$\{c_4, c_5\}$	6
7	$\{c_1, c_2, c_3, c_6\}$	$\{c_1\}$	7
8	$\{c_2, c_3, c_4, c_5, c_6\}$	$\{c_3\}$	7
9	$\{c_1, c_2, c_4, c_5, c_6\}$	$\{c_2\}$	8
10	$-$	$\{c_2\}$	8

Table 2: MaxHS algorithm execution with maximal cores.

Here Algorithm 5 must find nine minimum-cost hitting sets before an optimal MaxSAT solution to Equation 7 is found. To contrast this behavior, Table 3 shows an execution of the algorithm when MUSes are found instead. In this case, the same optimal solution is found after computing only three minimum-cost hitting sets.

Iteration	MUS	MCHS	MCHS cost
1	$\{c_1, c_2, c_3, c_4\}$	$\{c_4\}$	3
2	$\{c_1, c_2, c_5\}$	$\{c_4, c_5\}$	6
3	$\{c_2, c_3, c_6\}$	$\{c_2\}$	8
4	$-$	$\{c_2\}$	8

Table 3: MaxHS algorithm execution with minimal cores.

Every core (MUS) of Table 3 is a subset of multiple cores of Table 2. In essence, these smaller cores give stronger constraints for the hitting set problem, reducing the search space. Because the hitting set problem grows

with each iteration of the MaxHS algorithm, it becomes especially important to keep its size manageable on more difficult instances. Reducing core size achieves this both by limiting the number of cores that are required to find an optimal solution, and by producing smaller IP constraints in the MCHS formulation.

3.6.1 Re-refuting cores

A computationally inexpensive method of reducing core sizes in MaxHS, introduced in [37], is outlined in Algorithm 11. It exploits the fact that, depending on its state, a SAT solver might not produce the same core on consecutive calls even if the core is still present in the second instance. After finding a core κ in $\mathcal{F} \setminus H$, the solver can be invoked again to refute the unsatisfiable subformula $\mathcal{F}_h \cup \kappa$. This has the potential to produce a smaller core $\kappa' \subset \kappa$. If such a subset κ' is found, the process can then naturally be repeated on $\mathcal{F}_h \cup \kappa'$ to try to further reduce core size. The extent to which this technique can reduce cores is heavily dependent on the behavior of the SAT solver. The solver can be pushed to explore different areas of the search space in various ways, for example by removing learned clauses or changing variable decision heuristics (e.g., VSIDS scores [90]), to aid it in finding a different explanation for the unsatisfiability of $\mathcal{F}_h \cup \kappa$.

Algorithm 11 Reducing the size of a core by re-refutation.

```
1: function ReRefuteCore(F_h, κ)
2:     repeat
3:         s ← |κ|
4:         (sat, κ', τ) ← SolveSAT(F_h ∪ κ)
5:         if |κ'| < |κ| then κ ← κ'
6:     until s = |κ|
7:     return κ
```

3.6.2 Finding minimal cores

It is also possible to find cores which are minimal unsatisfiable subformulas (MUSes) in an attempt to maximize the potential benefit of smaller core size. The problem of finding a MUS M for a CNF formula $\mathcal{F}$ is D^{P}-hard, as it requires us to both show that $M \subseteq \mathcal{F}$ is unsatisfiable (in co-NP) and that $M \setminus \{C\}$ is satisfiable (in NP) for all $C \in M$. Various algorithms for MUS extraction have been proposed in recent years [81, 16, 14, 11, 91]. For MaxHS, even a simple "destructive" MUS algorithm[4], was shown to be effective in [38]. It was further suggested that the time spent on minimization is so small

[4] An algorithm which removes clauses from an initial MUS over-approximation as opposed to adding them to an under-approximation

Algorithm 12 A simple destructive MUS algorithm.

```
1: function MinimizeCore(F_h, κ)
2:     mus ← ∅
3:     for all C ∈ κ do
4:         κ ← κ \ {C}
5:         sat ← SolveSAT(F_h ∪ (mus ∪ κ))
6:         if not sat then
7:             mus ← mus ∪ {C}
8:     return mus
```

that more involved MUS algorithms are unlikely to have a significant impact on solving time. Algorithm 12 gives an example of this type of destructive MUS algorithm. Given a core κ for a partial MaxSAT instance $(\mathcal{F}_h, \mathcal{F}_s)$, we can try to reduce its size by checking the satisfiability of $\mathcal{F}_h \cup (\kappa \setminus \{C\})$ for some C in κ. If the formula is satisfiable, some MUS which is a subset of κ contains C. These clauses are gathered into a set mus by Algorithm 12. On the other hand, if $\mathcal{F}_h \cup (\kappa \setminus \{C\})$ is unsatisfiable, C is redundant and can be discarded.

Complete core minimization in this manner can dramatically increase the number of SAT solver calls. This has the potential to increase solving times, but it was empirically shown in [36] that this minimization has a significant overall positive effect. Beyond the positive effect on the IP constraints produced, the SAT solver calls produced by minimization are also typically on smaller subformulas $(\mathcal{F}_h \cup \kappa)$ than those used to derive the initial core $(\mathcal{F}_h \cup (\mathcal{F}_s \setminus H))$. Complete minimization can be detrimental if the cores of the MaxSAT instance are very large. In the worst case, even one of these "faster" SAT solver invocations can take exponential time. However, in practice such instances seem to be rare.

4 Re-implementing MaxHS

In order to more thoroughly understand MaxHS, the algorithm was re-implemented from scratch as part of this work. We call the resulting MaxSAT solver LMHS. In this section we compare it to existing solvers and use it to study the behavior of the MaxHS algorithm. LMHS also enabled the implementation of some extensions to the algorithm, which are covered in Section 5.

4.1 The LMHS Solver

Our re-implementation of MaxHS, LMHS, is implemented in C++ using standard library data structures. It can be used as a standalone solver, or as

a library through its C and C++ APIs. Like the original MaxHS solver[5], the re-implementation has been primarily developed using CPLEX [57] as an IP solver and MiniSat [44] as the SAT solver. However, we also accommodate the use of other SAT and IP solvers.

The implementation of LMHS differs in some key aspects from the original MaxHS solver. Perhaps the most important is its focus on modularity, allowing other IP or SAT solvers to be integrated into LMHS with little effort. We demonstrate this modularity by evaluating the performance of different solvers within the MaxHS algorithm in Section 4.3. We do not currently implement a model rotation [81] style minimal core extraction algorithm, as the simple destructive procedure of Algorithm 12 has proven to be very effective for LMHS. In LMHS we fix a value for every assumption variable on all SAT solver calls. We limit the generation of so-called non-core constraints to the equivalences of Section 3.5.2. We also do not explicitly encode the equivalence between a soft clause and its assumption variable in the formula. As noted in Section 3.5.2, this equivalence holds implicitly.

The LMHS solver was recently entered in the 2015 MaxSAT Evaluation[6], where it solved the most weighted partial MaxSAT instances in the crafted and industrial categories among non-portfolio solvers. The source code for the solver is made available at `https://github.com/psaikko/LMHS`.

4.2 Evaluating non-optimal hitting set strategies

Section 3.4 covered several methods for computing non-optimal hitting sets. In this section, we evaluate their effectiveness on a large set of benchmark instances. The strategies evaluated include the greedy hitting set algorithm, incremental hitting sets, and their combination.

The simplest non-optimal hitting set strategy "+1" creates a new hitting set by adding the most common clause (among all found cores) to an existing hitting set. The "greedy" strategy implements the greedy procedure of Algorithm 4, and "+1,g" implements Algorithm 9, combining "+1" and "greedy" in a two-phase approach. We also consider how search is affected by incrementally adding more clauses than necessary to the hitting set. The "+k%" strategies add k percent of the most common clauses (again, among all found cores) in a new core to an existing hitting set to create the new non-optimal hitting set. Intuitively, this could increase diversity among the cores found during each non-optimal phase by reducing overlap. In this sense, "+1" permits a maximal amount of overlap and "+100%" permits no overlap for cores found between optimal hitting sets. The "+100%" strategy causes an entirely *disjoint* set of cores to be found at each iteration so we refer to it as the "disjoint" strategy. Like "+1,g", "disjoint,g" implements a two-phase approach with the disjoint strategy and greedy algorithm.

[5] Available at `https://github.com/fbacchus/MaxHS`

[6] `http://www.maxsat.udl.cat/15/`

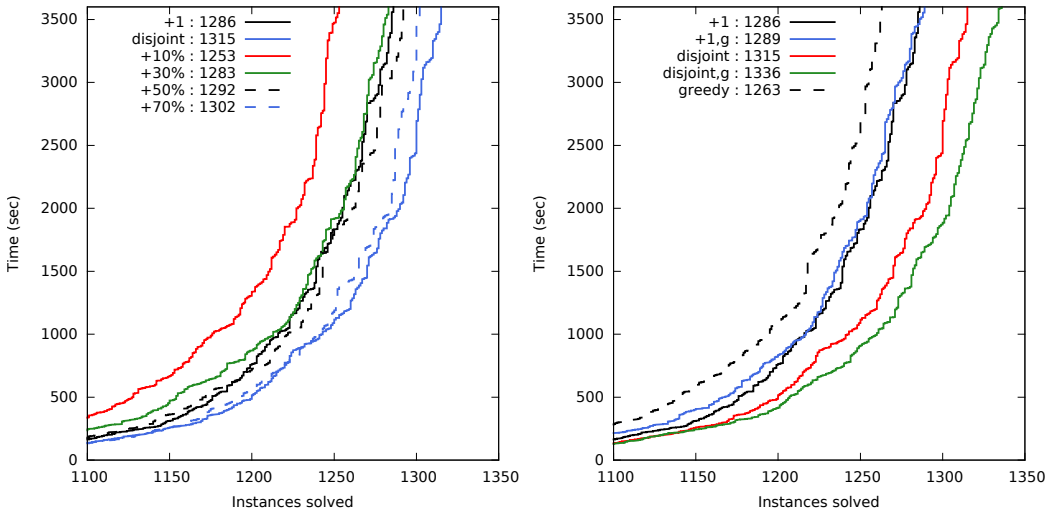


Figure 2: A comparison of (a) overlapping of cores and (b) greedy hitting set computations in the non-optimal hitting set strategies.

We evaluated the hitting set strategies on the entire set of 1710 partial and weighted partial MaxSAT benchmark instances from the crafted and industrial categories of the 2014 MaxSAT Evaluation[7]. All experiments reported on in this work (unless otherwise noted) were run on machines with 32 GB of memory and two Intel Xeon E5540 CPUs, with Ubuntu Linux 12.04. The computational resources for each instance were limited to 30 GB of memory and one hour (3600 seconds) of CPU time. The solver did not exceed the memory limit on any instances.

Figure 2 shows results in terms of per-instance timeouts over all 1710 instances. The vertical axis indicates the amount of time t in seconds given to the solver per instance and the horizontal axis indicates the number of instances solved. The plotted lines indicate, for each solver configuration, the number of instances that took less than t seconds to solve for every t from 0 seconds to the time limit of 3600 second. Figure 2a compares the single-phase non-optimal hitting set strategies and shows an interesting result. Decreasing the amount of overlap for cores found in a non-optimal hitting set phase, we see an uniform increase in the number of instances solved from the "+10%" strategy to the "disjoint" strategy. However, the "+1" strategy does not follow this trend, performing better than the "+10%" strategy while permitting more overlap between cores. Figure 2b shows the effect of adding the greedy hitting set algorithm as a second phase to the non-optimal hitting set computation. The difference between the "+1" and "+1,g" strategies

[7] `http://www.maxsat.udl.cat/14/benchmarks/index.html`

is small, but "disjoint,g" solves more instances within the time limit than either "disjoint" or "greedy".

To underline the practical benefit of reducing overlap, Figure 3 illustrates the difference between the "disjoint" and "+10%" strategies. We see in Figure 3a that, almost without exception, the "disjoint" strategy results in a very significant reduction in the number of cores needed to solve any particular instance. Intuitively, this can be seen as the result of increased diversity among the cores found in an iteration (i.e., an execution of the main loop of Algorithm 5) of the algorithm. On the other hand, Figure 3b shows that on average the "disjoint" strategy results in more iterations of the algorithm, and by extension, more IP solver invocations. However, as these IP solver invocations are on significantly smaller hitting set problem instances due to the reduced number of cores found, the net result is a large improvement in solver performance over the set of benchmark instances.

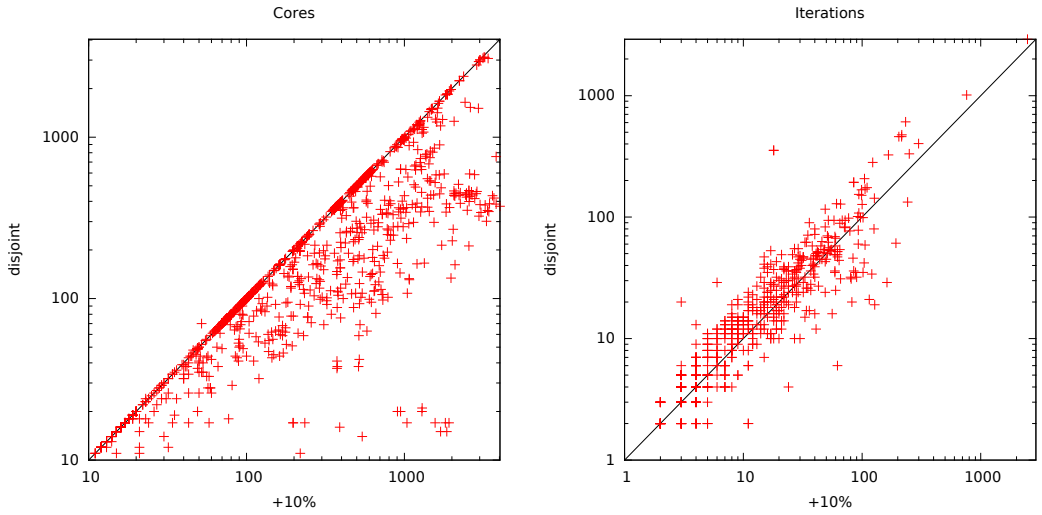
Figure 4 looks at the effect of the greedy phase by comparing the "disjoint" and "disjoint,g" strategies. Figure 4a shows that the greedy phase increases the number of cores needed to solve most instances. This negative effect on the solving process does not explain the improvement in the number of instances solved. Looking at the number of iterations needed to solve the benchmark instances instances, however, we see a clear improvement. The "disjoint,g" strategy results in much fewer iterations than the "disjoint" strategy. In this case, the reduced number of iterations significantly outweighs the somewhat larger number of cores required.

Figure 5 shows results for the comparison of non-optimal hitting set strategies by instance category. It is clear that "disjoint,g" is the best strategy, or among the best strategies, for every category shown. We also note that there is very little difference made by the choice of non-optimal hitting set strategy for crafted instances.

Table 4 shows results by instance family for some non-optimal hitting set strategies of interest. While the "disjoint,g" strategy solves the most instances overall, there are some families of instance for which it does not solve the most instances. This suggests that a heuristic for choosing the non-optimal hitting set strategy on a per-instance basis could result in a more well-rounded solver. More concretely, "disjoint,g" solves 1336 instances while a virtual best solver over the hitting set strategies of Table 4 (i.e., a solver which could choose the best strategy for instance) would solve 1359.

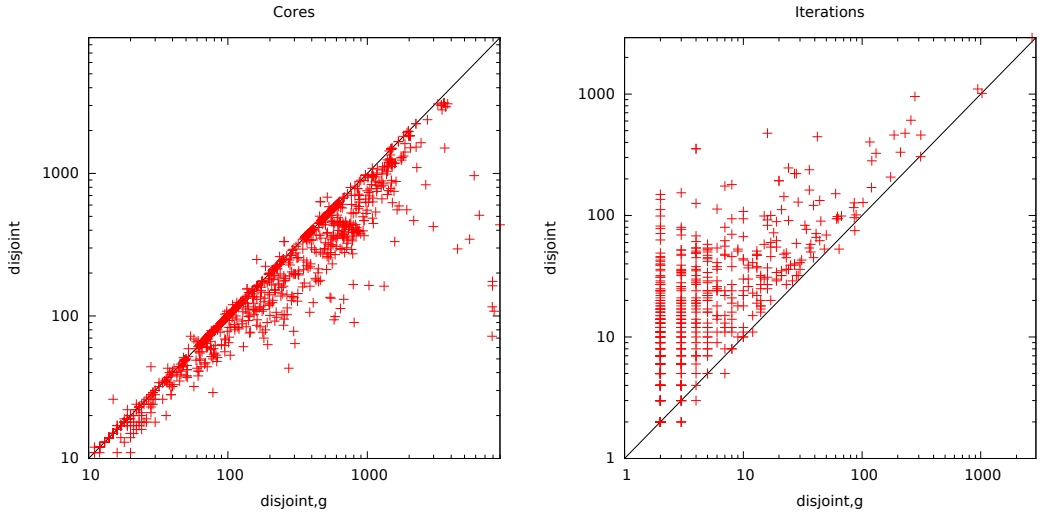
4.3 Impact of IP and SAT solvers

An important feature of our LMHS solver is its modularity with regard to its SAT and IP solver components. We have set a lightweight interface over which LMHS interacts with these solvers so that they can be easily replaced. This replacement process consists of implementing a small interface class which gives LMHS access to the required functionality of the SAT or IP



(a) (b)

Figure 3: The effect of reducing overlap on (a) the number of cores required to solve an instance and (b) the number of iterations until an optimal solution is found.



(a) (b)

Figure 4: The impact of the greedy phase on (a) the number of cores required to solve an instance and (b) the number of iterations until an optimal solution is found.

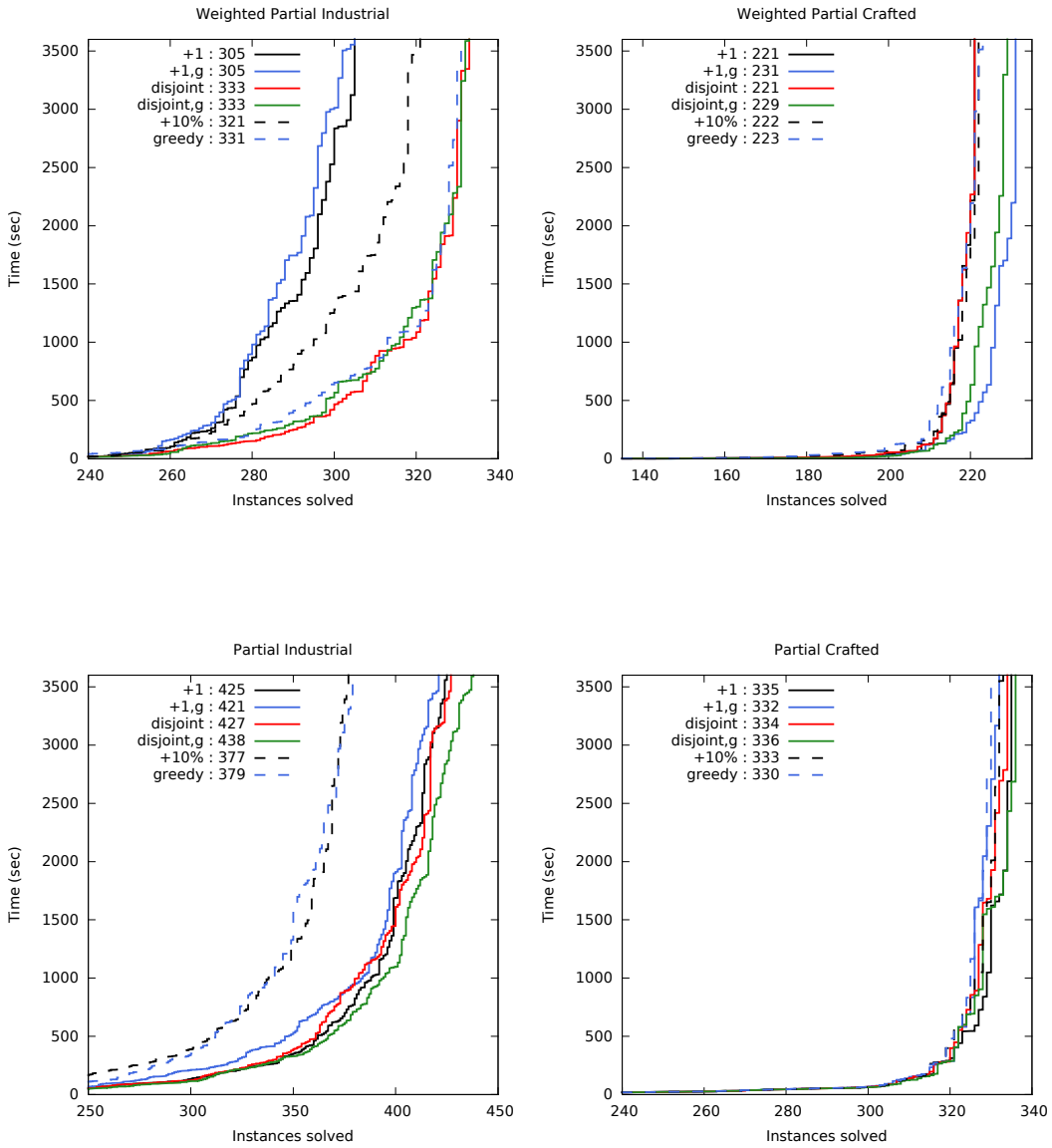
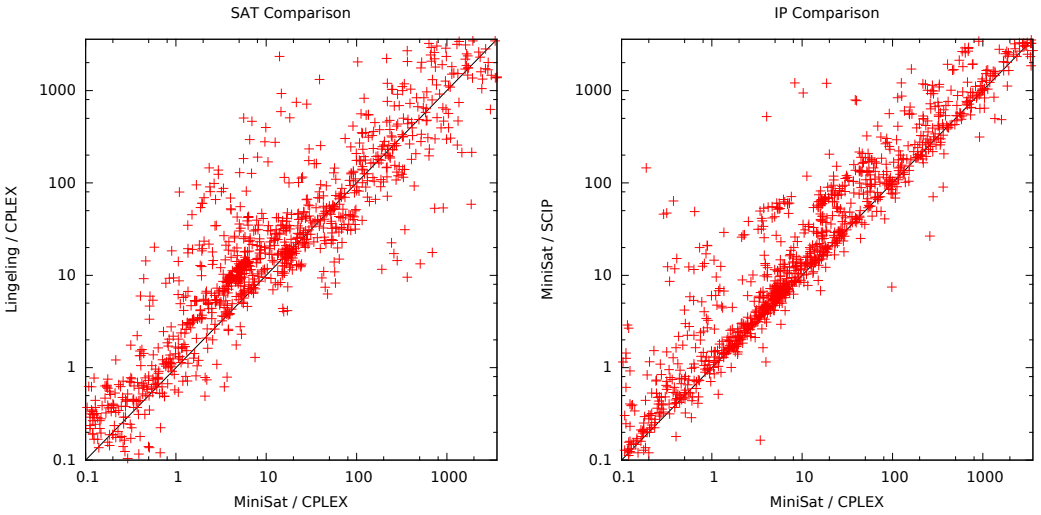


Figure 5: Comparison of results by instance category.

	+1		disjoint		disjoint,g		
Instance family	Solved	Time (s)	Solved	Time (s)	Solved	Time (s)	Total
Partial	760	966515	761	974010	**774**	**940325**	989
Crafted	335	**331275**	334	338202	**336**	332000	421
frb	15	43250	15	43248	15	**43246**	25
job-shop	3	**866**	3	881	3	877	3
maxclicque	134	94147	134	94148	134	**94125**	158
maxone	140	**753**	140	761	140	**753**	140
min-enc	8	**126614**	7	131658	**9**	127793	42
pseudo	4	12	4	12	4	12	4
reversi	31	47633	31	49494	31	**47194**	44
scheduling	0	18000	0	18000	0	18000	5
Industrial	425	635240	427	635808	**438**	**608325**	568
aes	2	**18622**	2	18625	2	18639	7
atcoss	22	57950	**23**	**57376**	**23**	57378	37
bcp	**144**	**191403**	131	238457	134	227798	188
circuit-trace-compaction	0	14400	0	14400	**1**	**11018**	4
close_solutions	24	106366	24	**106194**	24	106242	50
des	24	116809	**36**	**83490**	34	92749	50
haplotype-assembly	5	3656	5	3646	**6**	**3176**	6
hs-timetabling	1	3606	1	3606	1	3606	2
mbd	42	27313	42	**26266**	**43**	27369	46
packup-pms	40	327	40	**322**	40	386	40
pbo	65	2906	65	**1289**	65	2467	65
protein_ins	2	36017	2	36031	**4**	**30552**	12
tpr	54	55865	56	46106	**61**	**26945**	61
Weighted Partial	526	766935	554	663066	**562**	**643972**	721
Crafted	221	331703	221	332069	**229**	**309356**	310
CSG	10	98	10	**80**	10	81	10
auctions	40	1	40	1	40	1	40
frb	24	**43076**	24	43184	24	43162	34
min-enc	74	227	74	257	74	**197**	74
pseudo	3	32445	3	32439	3	**32427**	12
ramsey	1	50403	1	50403	1	50403	15
random-net	32	193	32	407	32	**161**	32
set-covering	36	**36052**	36	36082	36	36120	45
wmaxcut	1	169208	1	169216	**9**	**146804**	48
Industrial	305	435232	**333**	**330997**	**333**	334616	411
haplotyping-pedigrees	56	200961	**83**	101641	**83**	**101612**	100
hs-timetabling	2	45217	2	**45194**	**3**	45201	14
packup-wpms	100	1099	100	946	100	**673**	100
preference_planning	28	5310	28	5163	28	**4799**	29
timetabling	**7**	74114	**7**	**71274**	6	75063	26
upgradeability-problem	100	513	100	**511**	100	512	100
wcsp	12	108018	**13**	**106268**	**13**	106756	42

Table 4: Instances solved (time spent) for families of benchmark instances with different non-optimal hitting set techniques.



(a) (b)

Figure 6: A comparison of per-instance solving times for (a) SAT and (b) IP solvers within LMHS.

solver component. In this section we examine the performance of LMHS with different solver components. Specifically, we test the MiniSat 2.2 [44] and Lingeling ayv [23] SAT solvers, and the IBM CPLEX 12.6.0 [57] and SCIP 3.0.1 [2] IP solvers.

The LMHS solver was developed and tested largely using the MiniSat and CPLEX solvers. To illustrate the flexibility of this modular implementation, we chose two dissimilar SAT and IP solvers to test. The Lingeling SAT solver was chosen for its strong performance in recent SAT competitions[8], its inprocessing techniques, and the fact that unlike other successful solvers (e.g., glucose [8]) it is not based on MiniSat. The SCIP solver was chosen because it is non-commercial software.

Figure 6 gives a quick comparison of SAT and IP solver performance. Since the techniques and heuristics within LMHS have been chosen based on performance with MiniSat and CPLEX, it would be disingenuous to compare these components purely on the number of instances solved. For that reason, we compare solving times on a per-instance basis in Figure 6. We evaluated the SAT and IP solver components on the entire set of 1710 partial and weighted partial MaxSAT benchmark instances from the crafted and industrial categories of the 2014 MaxSAT Evaluation.

Figure 6a compares MiniSat and Lingeling. While using LMHS with Lingeling results in a significantly larger number of unsolved instances, there are also many instances which it solves faster as well as instances which

[8] `www.satcompetition.org`

are solved using Lingeling but not MiniSat. This suggests that it would be beneficial to investigate if Lingeling performs better in LMHS with different solving techniques or heuristics. It would also be interesting to identify if there are specific classes or families of instances which are more effectively solved using Lingeling. In contrast, Figure 6b shows that SCIP performs uniformly worse than CPLEX as the IP solver component.

4.4 Solver comparison

To conclude this section, we compare LMHS to other publicly available state-of-the-art MaxSAT solvers. We evaluate LMHS with the best non-optimal hitting set strategy of Section 4.2, "disjoint+g", using the best solver components of Section 4.3, MiniSat and CPLEX. Once again we used the set of 1710 partial and weighted partial MaxSAT benchmark instances from the crafted and industrial categories of the 2014 MaxSAT Evaluation.

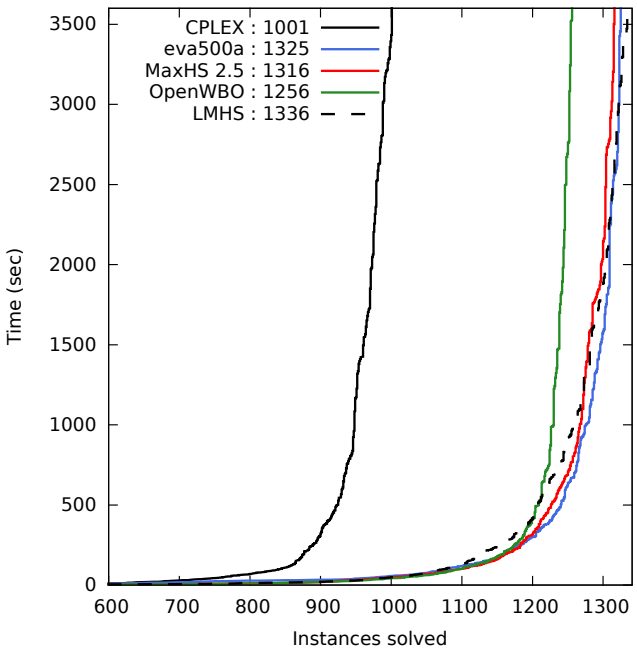


Figure 7: Solver performance on partial and weighted partial instances.

We use the CPLEX solver to test solving MaxSAT by reformulating instances as integer programming problems. The IP reformulation of Algorithm 4 is used. We also include eva500a [92], which solved the most industrial weighted partial MaxSAT problems in the 2014 evaluation. We compare also LMHS to MaxHS 2.5[9], the latest publicly available implementation of the MaxHS algorithm from the original authors at the time of

[9] `http://www.maxhs.org/`

writing. OpenWBO 1.3.1 [87], a top solver for industrial partial MaxSAT problems is also included in the comparison. We test OpenWBO in its default configuration, with MiniSat as the underlying SAT solver.

Figure 7 shows instances solved at different time intervals over all the test instances. It illustrates that LMHS is a robust general purpose MaxSAT solver for both partial and weighted partial MaxSAT problems, solving the most problems overall. We also show that LMHS outperforms MaxHS 2.5 without significant extensions to the algorithm. To further differentiate the solvers, we split the results by weighted and weighted partial instances as well as industrial and crafted instances in Figures 8 and 9.

We can make some interesting observations from Figure 8 and Figure 9. Firstly, while LMHS is competitive across the entire set of instances, it is especially effective for weighted partial MaxSAT problems. Another notable result is that while CPLEX performs poorly on the whole benchmark set, it performs especially well on instances in the crafted category.

5 Extensions

In this section we investigate various extensions to the MaxHS algorithm and empirically evaluate their effectiveness. Firstly, Section 5.1 looks at improving the efficiency of core extraction in MaxHS using a technique of assumption shuffling in the underlying SAT solver to produce multiple cores from a single conflict. Section 5.2 explores ways of parallelizing MaxHS using randomization of heuristic values and partitioning of soft clauses. Section 5.3 investigates the use of external propagators with the MaxHS algorithm, with graph acyclicity constraints as a case study. Section 5.4 gives an overview of the assumption variable reuse technique introduced in [20]. Finally, Section 5.6 introduces the incremental API of our solver, LMHS.

5.1 Fast core extraction through CDCL conflict analysis

In this section we investigate a simple technique for generating additional cores in a CDCL solver in polynomial time. By re-propagating assumptions in a different order after a conflict has been found, new sources of unsatisfiability could be found within the conflict produced by the CDCL search. We evaluate the effectiveness of this technique in isolation, as well as in conjunction with the other techniques discussed in Section 3.

The algorithm proceeds exactly as Algorithm 6 until the satisfiability of $\mathcal{F}$ under the assumptions A is determined. If $\mathcal{F}$ is satisfiable under A, the algorithm reports "Satisfiable" and returns a satisfying assignment. We note that like in Algorithm 6, UndoDecisions procedure cannot remove any assumptions A from D, so unsatisfiability has been determined if a conflict is found when $|D| = |A|$. At this point a core is found, and the algorithm deviates from Algorithm 6. Rather than immediately terminating, the initial

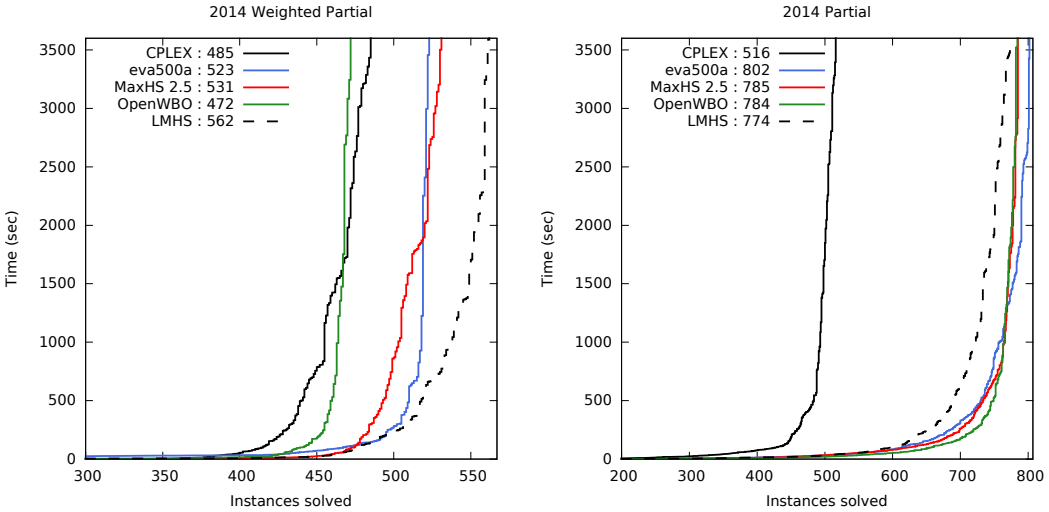


Figure 8: Results for benchmark instances on (a) weighted partial MaxSAT and (b) partial MaxSAT.

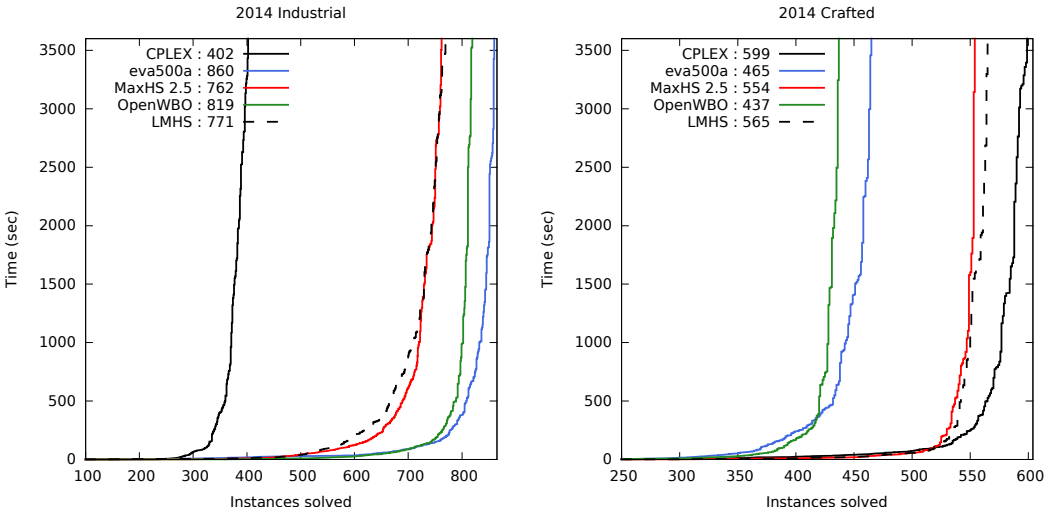


Figure 9: Results for benchmark instances on (a) industrial and (b) crafted instances.

Algorithm 13 Finding multiple cores with CDCL and assumption shuffling.

```
1: function ShuffleCDCL(F, A, k)
2:     D ← A
3:     K ← ∅
4:     τ ← ∅
5:     for i ∈ {1, ..., k} do
6:         while True do
7:             (c, τ) ← Propagate(F, D, τ)
8:             if c contains a conflict then
9:                 if |D| = |A| then                    ▷ conflict at base level
10:                    K ← K ∪ AnalyzeFinalConflict(c)
11:                    D ← RandomPermutation(A)
12:                    τ ← ∅
13:                else                                 ▷ learn a conflict clause
14:                    C ← AnalyzeConflict(c)
15:                    F ← F ∪ {C}
16:                    (D, τ) ← UndoDecisions(D, τ, C)   ▷ backjump
17:            else if τ(F) = 1 then
18:                return Satisfiable, τ
19:            else                                     ▷ make a new decision
20:                Choose a variable x unassigned by τ
21:                Make a decision for x: d ← {x = 0} or d ← {x = 1}
22:                D ← D ∪ {d}
23:    return RemoveRedundancy(K)
```

set of assumptions A is randomly permuted, or shuffled, and all propagated assignments are discarded.

At this point the clauses learned by the CDCL procedure are still in conflict with the set of assumptions A. This means that no further search, decisions, or backjumping will occur on the subsequent iterations of the loop of Line 5. On each of the iterations after the first, a conflict will be produced purely by propagation, in polynomial time. However, even with no further search, we may find a conflict which differs from the initial conflict produced. The new conflict can in turn cause AnalyzeFinalConflict to produce a new, distinct core. This is possible because the order in which assumptions or assignments are propagated within CDCL is significant. In essence, changing the order in which this propagation is performed can yield different cores which describe different conflicting aspects of the conflicting state of the CDCL procedure. This process is repeated k times (for some fixed k) to gather multiple cores. The random nature of this approach means that many of the cores generated may be redundant, so we remove duplicate cores and supersets of cores within $\mathcal{K}$ as a final step. Further refinement can

also be done here, for example to remove cores which are too large or too similar to other cores in $\mathcal{K}$.

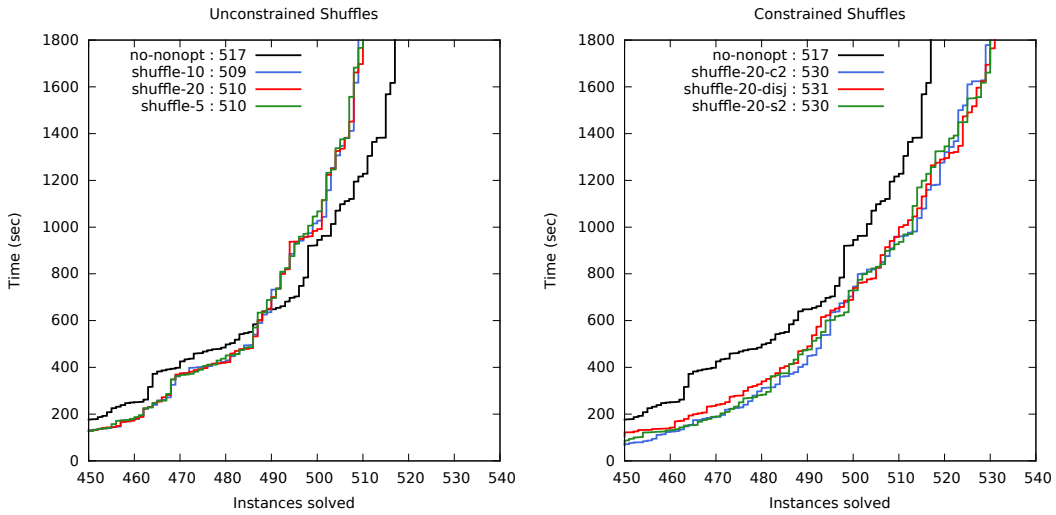
5.1.1 Experiments

We performed an experimental evaluation of the effectiveness of the assumption shuffling technique for generating cores within LMHS. Shuffling and non-optimal hitting sets fill a similar role in providing a relatively inexpensive way to generate additional cores. To isolate the effect of shuffling, we first compare it (with various refinements) to MaxHS without non-optimal hitting sets. We then compare the best shuffling technique to the best non-optimal hitting set strategy (as determined in Section 4.2) and evaluate the degree to which they work in conjunction with each other. Experiments were run on only the weighted partial MaxSAT benchmark set, with a 30 minute (1800s) time limit.

Figure 10 compares different techniques for assumption shuffling. The "no-nonopt" test shows the performance of the LMHS solver without any non-optimal hitting set strategies. On the left, Figure 10a looks at results for unconstrained shuffling. The "shuffle-k" techniques perform k shuffles and use all unique cores found through shuffling (again, without non-optimal hitting sets). The cores found through shuffling will naturally tend to be similar as they are the product of some single conflict in the SAT solver. Figure 10a also shows that no significant overhead is added by the shuffling process, as the results for 5, 10, and 20 shuffles are nearly identical. These results would then seem to show that the MaxHS algorithm does not benefit from large amounts of similar cores, even if they are found at no extra cost.

Figure 10b evaluates different ways of limiting the number of found cores used. We examine three of the best results: "shuffle-20-c2" performs 20 shuffles but only uses the two smallest cores, "shuffle-20-s2" likewise performs 20 shuffles but takes only the smallest core and any cores of size 2 or smaller, and "shuffle-20-disj" takes only a disjoint set of cores. Each of these constrained shuffling techniques clearly improves over the baseline result of "no-nonopt".

By Figure 10b alone, it would appear that the constrained shuffling techniques are nearly identical. However, the solving times of individual instances in Figure 10c show that the "shuffle-20-disj" and "shuffle-20-c2" can exhibit different behavior. Finally, Figure 10d looks at the effect of shuffling in conjunction with non-optimal hitting set techniques. We compare LMHS without non-optimal hitting sets ("no-nonopt") to LMHS ("LMHS"), LMHS with shuffling but without non-optimal hitting sets ("shuffle"), and LMHS with shuffling ("LMHS+shuffle"). Although shuffling alone improves the performance of LMHS, we do not see any significant improvement when it is used in conjunction with non-optimal hitting sets. However, assumption shuffling is not a MaxHS or MaxSAT specific technique and could result in



(a) (b) (c) (d)

Figure 10: A comparison of (a) unconstrained and (b) constrained shuffling techniques, per-instance solving times (c) for two similar constrained shuffling techniques, and (d) a look into the overall effect of shuffling with LMHS.

improvements for core extraction in other contexts.

5.2 Parallelizing MaxHS

MaxHS also presents some new possibilities for parallelizing MaxSAT solving. A sequence of SAT and IP solver calls gathers a set of cores $\mathcal{K}$, which yields an optimal solution. If we could efficiently perform this core extraction in parallel on k threads, solving time could decrease by a factor of k on families of instances for which MaxHS uses most of its time in SAT solving. Such instances seem to exist, and methods such as non-optimal hitting set strategies and core minimization further reduce the portion of time used for IP solving [39].

We would like to be able to parallelize this core extraction by simply invoking k instances of the SAT solver on $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$ in parallel at each iteration. However, this is potentially inefficient as instances of a SAT solver invoked on the same formula are likely to find identical cores. It may be possible to reduce the number of these duplicate cores found by heuristic methods such as initializing each solver with different VSIDS scores to guide them to different areas of the search space.

In this section we explore parallelizing MaxHS in two ways. We test a simple parallelization method which runs k instances of a SAT solver in parallel with randomized VSIDS scores. We also experiment with parallelization by partitioning the soft clauses of a partial MaxSAT instance.

5.2.1 Partitioning

One way of eliminating the issue of duplicate cores involves partitioning $\mathcal{F}_s$ into sets of soft clauses $P_1, \ldots, P_k$. The SAT solvers can then be invoked in parallel on the subformulas $\mathcal{F}_h \cup (P_i \setminus H)$. If the partitions P_i are non-overlapping, any cores found in the subformulas $\mathcal{F}_h \cup (P_i \setminus H)$ are also guaranteed to be non-overlapping for a given iteration. The hitting set H prevents us from finding duplicate cores between iterations.

Partitioning of soft clauses in MaxSAT has been recently explored in other contexts. Some examples of partitioning strategies include weight-based partitioning, hypergraph-based partitioning, and resolution-based partitioning. A simple weight-based partitioning strategy [86] would group together clauses with similar weights. Hypergraph-based partitioning [86] interprets the soft clauses as a hypergraph with clauses as nodes and variables as hyperedges, and applies some standard hypergraph partitioning algorithm. The recent resolution-based partitioning technique has been shown to improve the performance of a SAT-based MaxSAT solver [93] even without parallelization. Besides a partitioning strategy, we must also consider the relative sizes of the partitions, whether or not they overlap, and how to create unique partitionings at each iteration.

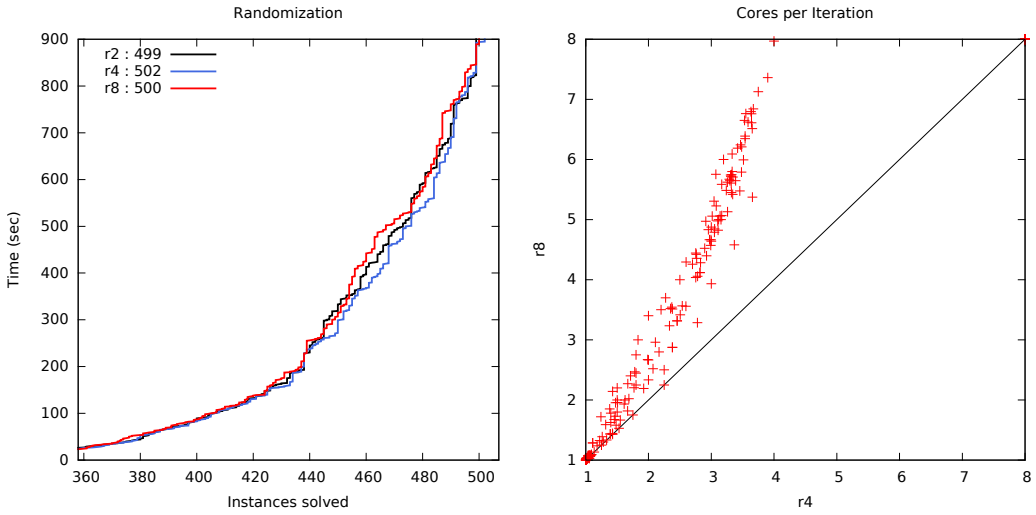
Algorithm 14 Parallelizing core extraction in MaxHS.

```
1:  function ParallelMaxHS(F_h, F_s, c, k)
2:      K ← ∅, H ← ∅
3:      F_C ← ∅, C ← ∞
4:      while true do
5:          P_1, ..., P_k ← Partition(F_s \ F_C)
6:          for i ∈ {1, ..., k} do in parallel
7:              F_Si ← (F_C ∪ P_i) \ H
8:              (sat_i, κ_i, τ_i) ← SolveSAT(F_h ∪ F_Si)
9:              if sat_i and c(τ_i) < C then
10:                 F_C ← F_Si
11:                 C ← c(τ_i)
12:             else
13:                 K ← K ∪ {κ_i}
14:         if no new cores were found then
15:             (sat, κ, τ) ← SolveSAT(F_h ∪ (F_S \ H))
16:             if not sat then K ← K ∪ {κ}
17:             else return τ
18:         H ← SolveMCHS(K, c)
```

We would like for a partitioning strategy to create a large number of unsatisfiable partitions. If a partition is unsatisfiable, it will yield a core which refines the next hitting set, and the algorithm will make progress. Larger partitions are more likely to contain cores, but the choice of partitioning strategy could be important as well. We would also like to gain some knowledge from partitions which are satisfiable, even if they do not directly contribute any cores.

We must also guarantee that an optimal solution will be found. For example, if we partition m clauses into k non-overlapping partitions of equal size, any optimal solution that satisfies more than m/k soft clauses will not be found. To avoid this, we could decrease k over time such that eventually only a single partition which contains all of $\mathcal{F}_s$ remains, but this has the disadvantage of reducing the amount of parallelization. The partition-based solver of [93] gradually merges the initial partitions together so that eventually the entire formula is solved, but intuitively this approach too would reduce the impact of parallelization.

With these considerations, we propose Algorithm 14, which creates overlapping partitions which do not share any unsatisfiable cores. Algorithm 14 parallelizes the core extraction of MaxHS using partitioning. Line 3 initializes the satisfiable subset of soft clauses $\mathcal{F}_\mathrm{C}$ its cost C. A potentially interesting modification to the algorithm is to reset $\mathcal{F}_\mathrm{C}$ before every iteration. At the start of the main loop, Line 5, the clauses $\mathcal{F}_s \setminus \mathcal{F}_\mathrm{C}$ are split into into k



(a) (b)

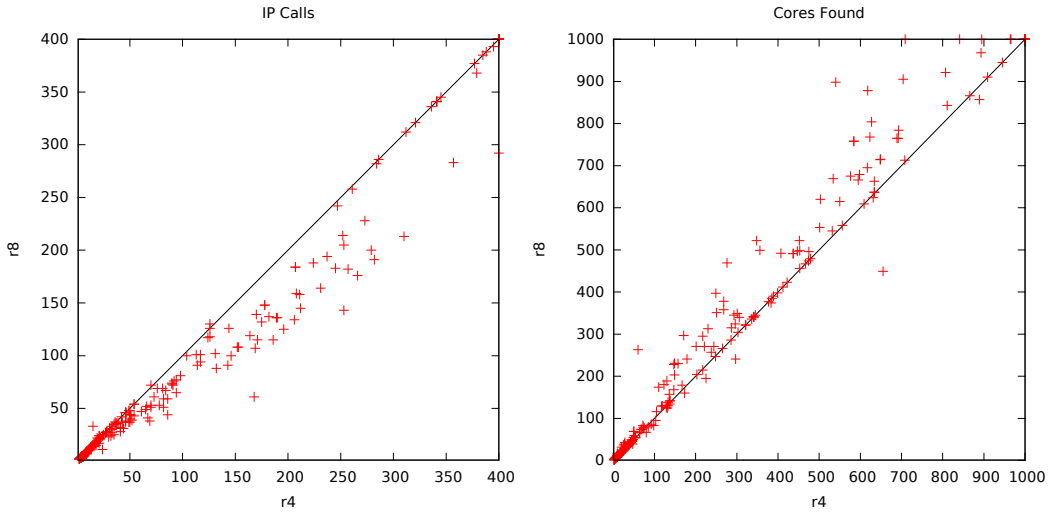
Figure 11: Comparing (a) the number of instances solved by parallel solvers "rk" and (b) the average number of unique cores found per iteration.

non-overlapping partitions. Instances of a SAT solver are then invoked in parallel (Line 7) on the subformulas $\mathcal{F}_h \cup ((\mathcal{F}_C \cup P_i) \setminus H)$. The overlap in the subformulas is restricted to a set of soft clauses, $\mathcal{F}_C$, which has been identified as satisfiable, eliminating the possibility of two solvers finding duplicate cores on the same iteration. Line 9 updates the satisfiable set of soft clauses $\mathcal{F}_C$ and its cost C. As $\mathcal{F}_C$ grows, we approach the situation of invoking SAT solvers in parallel on the entire formula. When all partitions for an iteration are satisfiable, Line 14 checks the satisfiability of $\mathcal{F}_h \cup (\mathcal{F}_s \setminus H)$, the MaxHS termination condition. If the check fails, a new core is found and the algorithm resumes. Otherwise an optimal solution has been found.

5.2.2 Experiments

We perform an experimental evaluation of the parallelization techniques explored in Section 5.2. Experiments were performed on the set of 2014 MaxSAT evaluation weighted partial instances with a time limit of 15 minutes (900 seconds). As before, the experiments were run on machines with 32 GB of memory and two Intel Xeon E5540 CPUs, for a total of eight physical CPU cores per machine.

We first examine the performance of the randomized approach in Figure 11. Here "rk" denotes that k instances of the SAT solver are run in parallel, each with randomized VSIDS variable decision heuristic scores. In Figure 11a we see no visible improvement by adding parallel solvers using this heuristic value randomization. However, this lack of improvement cannot be attributed to a



(a) (b)

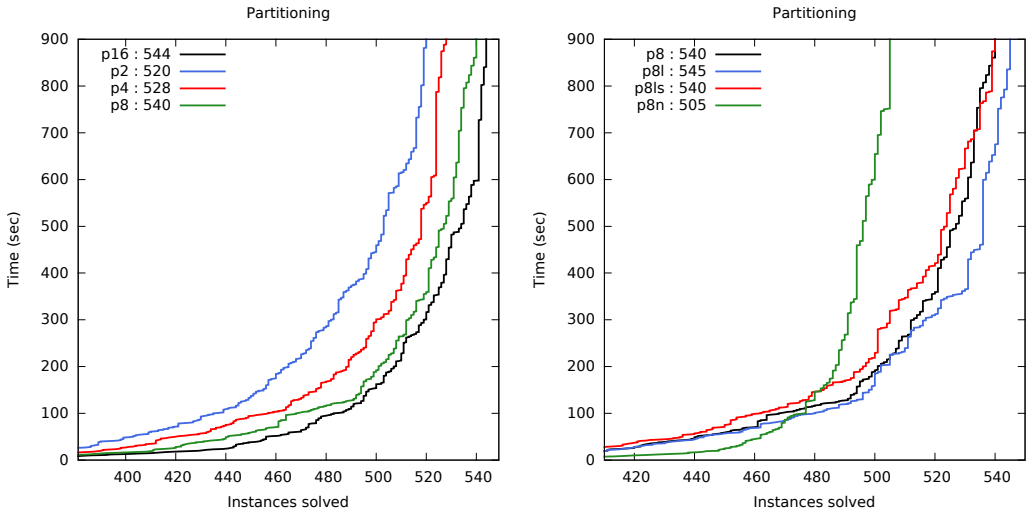
Figure 12: Comparing the (a) the number of iterations and (b) the number of cores required to solve instances for "r4" and "r8".

failure of the randomization to produce different cores on the concurrently run SAT solvers. Figure 11b shows the average number of distinct cores found per iteration for the test instances. Doubling the number of concurrent solvers from four to eight also roughly doubles the number of cores found per iteration.

Figure 12 provides a partial explanation for the results. While the number of cores found per iteration tends to double from "r4" to "r8", Figure 12a shows that the number of iterations (IP solver calls) is not correspondingly cut in half. It follows that there must be an increase in the number of total cores found, as seen in Figure 12b. This combination of fewer iterations with a larger set of cores seems to result in virtually identical performance overall for "r4" and "r8". It remains an interesting question to characterize the underlying cause of the MaxHS algorithm's apparent "resistance" to this method of parallelization.

Next, we evaluate the effectiveness of Algorithm 14. Figure 13a shows the change in the number of instances solved as we increase the number k of partitions and concurrently run solvers. Here "pk" indicates that k solvers are run on k partitions. These tests are run without an initial disjoint phase or non-optimal hitting set strategies to better isolate the effect of partitioning and parallelization. A very simple partitioning scheme in which soft clauses are evenly divided into k partitions based on their order in the instance file is used. Furthermore, we do not run the IP solver with multiple threads.

We see improvement in the performance of the algorithm as k increases.



(a) (b)

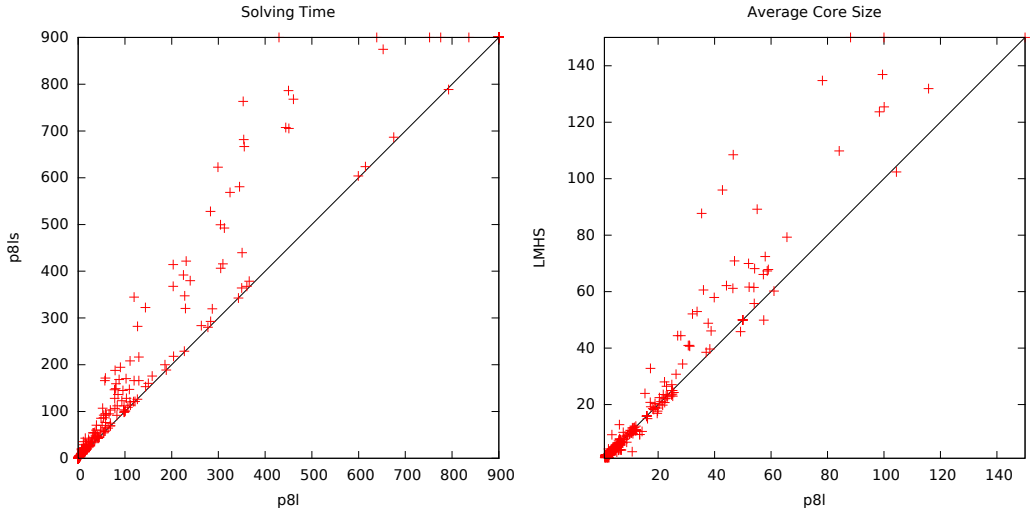
Figure 13: An overview of the performance of parallelization by partitioning.

Interestingly, this effect persists between "p8" and "p16" although the tests were run on machines with only eight physical CPU cores. This suggests that the partitioning itself is at least in part responsible for these improvements, as running 16 solvers concurrently on eight CPU cores does not seem likely to have a positive effect on solving times.

Figure 13b shows several additional results. We first note that a naïve implementation "p8n" of non-optimal hitting sets—in which each concurrent SAT solver performs its own non-optimal phase—is detrimental to the performance of Algorithm 14. We also evaluate "p8l", a variant of Algorithm 14 which picks a new $\mathcal{F}_C$ on every iteration. This results in a small, but consistent, improvement over "p8". Finally, Figure 13b compares "p8l" with its sequential execution "p8ls". Here we see that the effect of parallelization is relatively small, which again suggests that even the simple partitioning of soft clauses has a beneficial effect.

To further compare "p8l" and "p8ls", Figure 14a shows solving times on individual instances. Here we see that although the number of additional instances solved is fairly small, there is a consistent improvement in solving time, especially among instances taking longer than 200 seconds to solve. Figure 14b motivates the effectiveness of partitioning, comparing the average core size solved instances between LMHS and "p8l". We see that especially among instances with large cores on average, "p8l" finds cores which are significantly smaller than those found by LMHS.

Figure 15 compares Algorithm 14 to LMHS with ("LMHS") and without ("no-nonopt") non-optimal hitting sets. We see an overall improvement on



(a) (b)

Figure 14: Comparing (a) the solving time of "p8l" to its sequential execution and (b) the average core size on instances for "p8l" and LMHS.

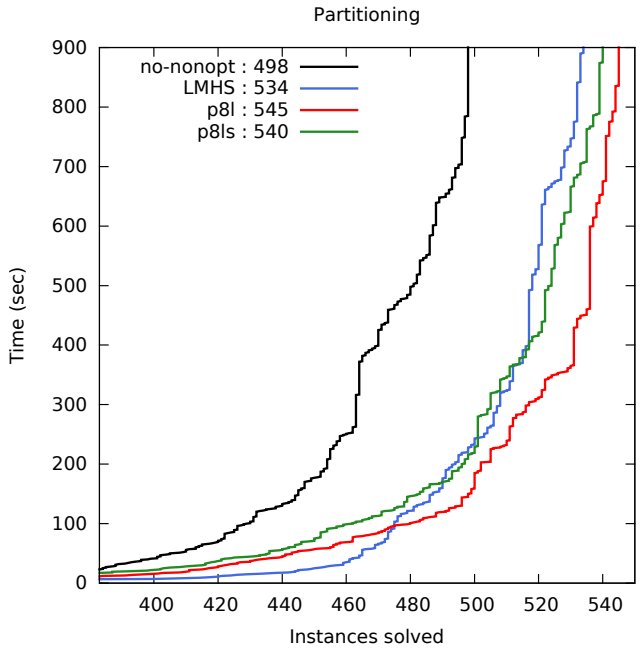


Figure 15: Evaluating Algorithm 14 by comparison to LMHS.

LMHS by both "p8l" and its sequential execution "p8ls". We note that this improvement is achieved without non-optimal hitting sets or a disjoint phase. Comparing "p8l" to "no-nonopt", we see a significant increase in the number of instances solved. This suggests that further improvements could be seen by successfully integrating non-optimal hitting set strategies with Algorithm 14. We also note that these tests used a very rudimentary partitioning scheme. Further investigation of more sophisticated partitioning strategies seems likely to yield results with MaxHS.

5.3 External propagators

The CDCL procedure (recall Algorithm 1) makes use of unit propagation in its inference, but can be extended to utilize other types of constraint propagation as well. This propagation need not necessarily be implemented on the CNF level. In this section we investigate propagation based on external constraints with the MaxHS algorithm.

As a motivating example, consider cardinality constraints of the form $x_1 + \cdots + x_n \leq k$. Many different CNF encodings for these constraints exist [6, 10, 5, 30, 45, 80] but even the most concise encodings require a linear number of additional clauses and variables [104]. In contrast to these encodings, the cardinality constraint could be enforced by a separate propagator within a CDCL solver. Such a propagator could be implemented as a counter which is incremented whenever $x_i = 1$ is assigned for some i and decremented when these assignments are undone by backtracking. Checking whether $x_1 + \cdots + x_n \leq k$ holds can then be done by simply checking the counter. If $x_1 + \cdots + x_n = k - 1$, then $x_i = 0$ can be propagated for all unassigned variables in $x_1, \ldots, x_n$. Propagators of this nature can implicitly fill the role of a large set of clauses [96], and have found use in satisfiability modulo theories (SMT) solvers [102].

Any external propagator which is valid for SAT is valid for MaxHS. However, it is not clear whether use of such propagators is useful within MaxHS or MaxSAT solvers in general as the CNF formulas of MaxSAT instances often tend to be (due to the difficulty of the optimization problem) much smaller than those of SAT instances. MaxHS in particular solves a sequence of SAT problems that tend to become easier to solve with each iteration, which could reduce the impact of these propagators. It is also unclear how the performance of clause learning is affected in practice by the coupling of CDCL and external propagators. This section examines graph acyclicity as a case study for implementing external propagators within LMHS.

Algorithm 15 A propagator for acyclicity constraints [49].

function AcyclicityPropagator($x, f, (V, E)$)
  if x does not enable an edge **then return**
  $(v_a, v_b) \leftarrow f^{-1}(x)$
  let E' be the set of edges e for which $f(e) = 1$.
  let G' be the subgraph of (V, E) induced by E'.
  for each node v on a forward traversal of G' from v_b **do**
    mark v
    if $v = v_a$ **then**
      let $e_1, \ldots, e_k$ be the edges on path $v_b \rightarrow v_a$
      $C \leftarrow f(e_1) \vee \cdots \vee f(e_k)$
      return with conflict clause C
  for each node v'_b on a backward traversal of G' from v_b **do**
    for each edge $e = (v'_a, v'_b)$ such that v'_a is marked **do**
      $x' \leftarrow f(e)$
      if x' is unassigned **then**
        let E_f be the set of edges on the path $v_b \rightarrow v'_a$
        let E_b be the set of edges on the path $v'_b \rightarrow v_a$
        $C \leftarrow \neg x \vee \neg x' \vee \bigvee_{e \in E_f \cup E_b} \neg f(e)$
        based on C, propagate $\neg x'$

5.3.1 Acyclicity constraint propagation

We follow [49] in implementing acyclicity propagation for a SAT solver to study the use of external propagators in conjunction with MaxHS. Acyclicity propagation can be used if the problem to be solved is either directly a graph problem, or has some underlying graph structure on which an acyclicity condition must hold. Acyclicity constraint propagation also has applications beyond graph acyclicity. For example, a set of variables $\{x_1, \ldots, x_n\}$ has a partial order $\leq$ if and only if the graph with nodes $x_1, \ldots, x_n$ and edges $\{(x_i, x_j) : x_i \leq x_j\}$ is acyclic. Potential applications include problems of computing optimal tree or DAG structures and problems for which linear orderings must be enforced, e.g., the NP-complete problem of finding an optimal variable ordering for ordered binary decision diagrams [25].

We consider acyclicity in terms of a directed graph structure $G = (V, E)$ and a function $f : E \rightarrow X$ that maps each edge to some Boolean variable in the CNF formula. The graph G and mapping f must be given to the SAT solver as supplemental information. This information is used by the acyclicity constraint propagator (Algorithm 15) to prevent variable assignments which correspond to cyclic graphs.

Algorithm 15 is based on two traversals of the graph G. Whenever a literal x corresponding to a graph edge $e_x = (v_a, v_b)$ is propagated, the algorithm checks if this edge causes a cycle given the current state of the graph and

whether any further assignments can be propagated. The algorithm first traverses the graph forward from v_b (Line 6), detecting any cycles caused by the current variable assignment. If a cycle $e_1, \ldots, e_k, e_1$ is found, a conflict clause consisting of the variables $f(e_1), \ldots, f(e_k)$ of the edges of the cycle is returned. The second traversal (Line 12) searches the graph backwards (i.e., from a node v_b to a node v_a along an edge (v_a, v_b)), detecting unassigned variables which could form a cycle. If an unassigned variable x' which could complete a cycle is found, we can propagate the assignment $x' = 0$. This propagation is done based on the implicit clause $\neg x \vee \neg x' \vee \bigvee_{e \in E_f \cup E_b} \neg f(e)$ which would prevent the cycle.

5.3.2 Experiments

To test the effectiveness of the acyclicity constraint propagation, we implemented it within MiniSat. This modified MiniSat was used within LMHS to enable us to solve MaxSAT problems with external acyclicity constraints. We applied this solver to the problem of bounded treewidth Bayesian network structure learning (BTW-BNSL) [46]. We examine Bayesian network structure learning in more detail in Section 6. A MaxSAT formulation for BTW-BNSL was recently introduced [19]. We focus on this application as the MaxSAT formulation, shown to be the best approach for optimally solving BTW-BNSL problems [19], has two clear applications for acyclicity propagation.

The exact MaxSAT formulation for BTW-BNSL is beyond the scope of this work (see [19] for details), but we give a quick summary of the uses of acyclicity propagation we experiment with. The straightforward application of acyclicity propagation in the MaxSAT BTW-BNSL formulation is in enforcing the acyclicity of the resulting DAG. The more indirect application involves the treewidth encoding. More specifically, the treewidth encoding requires enforcing a linear ordering for a set of variables. Acyclicity is enforced in [19] with a level-based CNF encoding, which creates a polynomial number of clauses and variables (in the size of this set of variables), which makes it desirable to try to replace this encoding with an external propagator.

We experiment both with replacing the DAG structure acyclicity encoding (configurations labeled "DAG") and replacing the treewidth linear ordering encoding (configurations labeled "TW") with the external acyclicity propagation. We also test Algorithm 15 in LMHS both with and without the second, backward graph traversal. Configurations which use only the first traversal are labeled "Inc", as they only detect inconsistencies. Configurations which also perform propagation using the backward graph traversal are labeled "Back"[10]. As benchmark instances we use data sets used in [19] as well as standard BNSL benchmarks with treewidth restrictions for a total

[10]We use this labeling for consistency with [49].

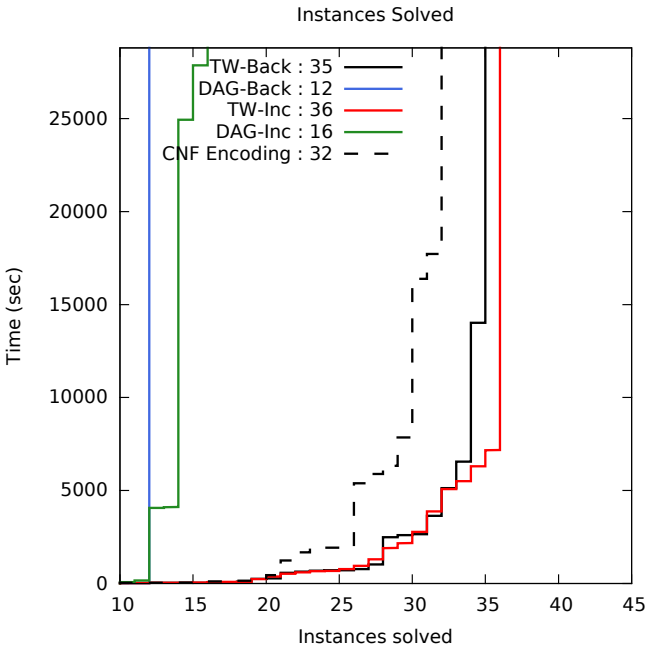
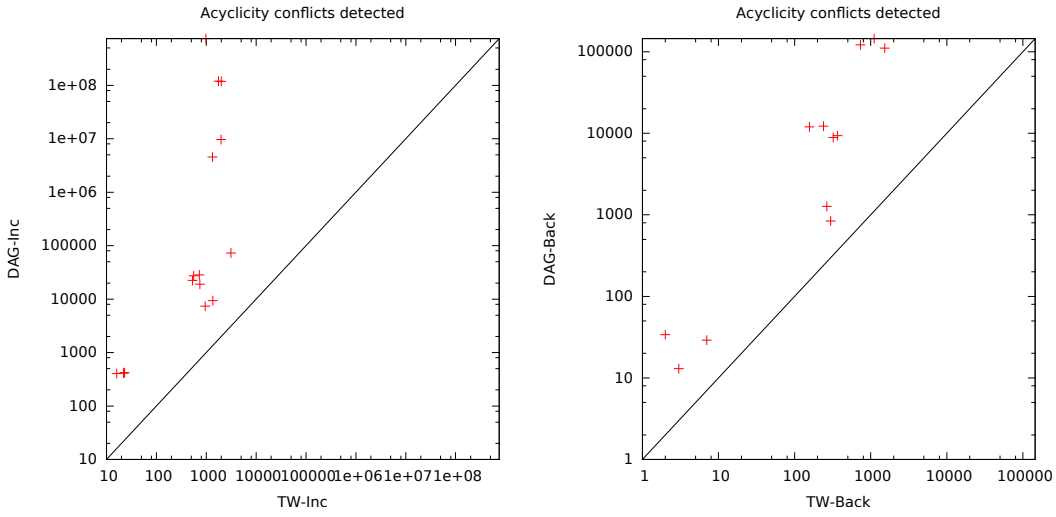


Figure 16: A comparison of acyclicity propagation techniques.



(a) (b)

Figure 17: Comparing instances solved by both "TW" and "DAG" with (a) "Inc" and (b) "Back" propagators.

	CNF Encoding			TW-Inc		
Instance	TW2	TW3	TW4	TW2	TW3	TW4
Abalone	136.5	448.9	175.1	84.8	233.7	370.7
Heart	-	-	6329.3	5079.6	5507.9	6301.6
Hepatitis	-	-	17725.1	2772.9	2169.9	1907.8
Horse	1240.1	1678.1	1928.2	659.3	1300.3	677.1
Wine	149.0	42.8	48.7	40.5	48.9	35.6
Zoo	35.1	115.3	67.8	36.0	82.1	77.3
adult15N	1997.3	16384.3	7855.8	953.3	7171.7	3871.2
asia_10000	4.8	2.6	2.8	2.1	2.0	3.4
asia_1000	0.6	0.9	1.0	0.8	1.0	0.6
asia_100	0.1	0.1	0.0	0.1	0.1	0.1
hailfinder_1000	5384.4	1874.5	5889.0	519.6	595.4	779.0
hailfinder_100	22.7	32.5	33.1	14.7	22.0	21.9

Table 5: Solving times (in seconds) for solved BTW-BNSL instances.

of 63 test instances (21 instances each with treewidth bounds of 2, 3, or 4). We obtained the benchmarks directly from the authors of [19].

Figure 16 shows the number of instances solved at different per-instance timeouts. With our proof-of-concept implementation, we see an improvement over the CNF encoding with acyclicity propagation on the treewidth constraint. Interestingly, the "Back" configuration solves fewer instances than the "Inc" configuration with both "TW" and "DAG" propagation.

When applying the external propagator to the DAG structure, we see significantly worse results. Figure 17 provides some insight into this behavior. For both the "Ìnc" and "Back" configurations, we see a significant increase in the number of conflicts detected during solving. A potential cause for this could be that we detect some smaller set of conflicts multiple times across a number of SAT solver calls. LMHS forgets learned clauses between SAT solver calls by default, but it may be beneficial to make an exception for clauses learned from acyclicity conflicts. We could also see better results from the "Back" configurations by explicitly storing the implicit clauses of Algorithm 15 Line 18.

Table 5 gives more detailed results for the acyclicity propagation. We show individual solving times for each instance, for different treewidths, where TWk corresponds to a treewidth limit of k. Fields marked "-" denote instances on which the solver exceeds the eight hour time limit. We see that LMHS using acyclicity propagation with the treewidth constraint ("TW-Inc") solves a superset of the instances solved with the pure CNF encoding, and typically solves these instances faster.

5.4 Reusing assumption variables

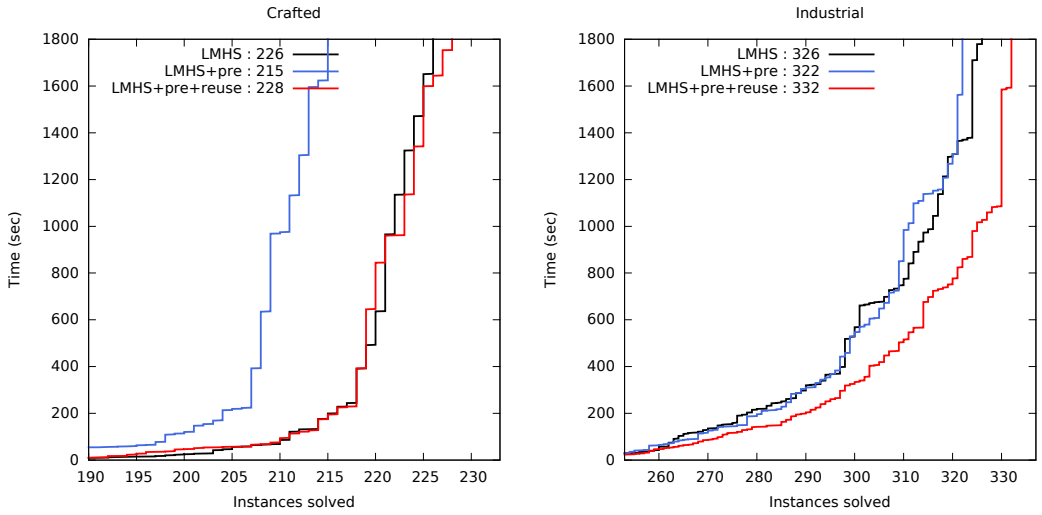
Although the MaxHS algorithm does increase the size of the CNF formula during the solving process, some overhead is introduced by the assumption variables added for each soft clause. In some cases, it is possible to reuse variables already present in the instance to reduce this overhead. A simple approach would identify a hard clause $(C \vee x)$ and a soft clause $(\neg x)$, where the variable x appears only in these clauses. Here we can discard both clauses, replacing them with a single soft clause $(C \vee x)$, and reusing x as the assumption variable. However, such a pair of clauses is likely rare in practice, as it is semantically identical to a single soft clause C.

A key observation [20] is that the MaxHS algorithm is sound even in cases where an assumption variable is shared between clauses or a clause contains more than one assumption variable. This means that any literal $l \in \{x, \neg x\}$ which occurs only in a single unit soft clause $(\neg l)$ and some hard clauses $(C_1 \vee l), \ldots, (C_n \vee l)$ of an instance can be reused as an assumption. In other words, with the exception of the single unit soft clause, occurrences of x in the instance must have the same polarity. Clauses of this type occur naturally in, e.g., encodings of group MaxSAT [55]. As we showed in [21], such clauses are in fact common in standard benchmark instances.

A method for applying SAT preprocessing techniques to MaxSAT [17] also produces variables which are suitable for reuse. Many SAT preprocessing techniques, such as bounded variable elimination [42], are not sound for MaxSAT on their own [17]. However, they can be made sound by introducing a layer of auxiliary variables and forbidding their removal [17, 15]. Concretely, prior to applying preprocessing, a new variable x_i is introduced for each each soft clause C_i. The original soft clause replaced with a hard clause $(C_i \vee x_i)$ and soft clause $(\neg x_i)$, with the restriction that $(\neg x_i)$ may not be eliminated from the formula. This is essentially the reverse of the simple assumption variable detection procedure discussed earlier. Crucially, each of these new $x_i$variables can be used as an assumption variable even *after* applying preprocessing. With this observation, MaxHS need not add any new assumption variables for preprocessed instances.

5.5 Experiments

We experimentally confirm the effectiveness of reusing assumption variables in conjunction with preprocessing in LMHS. For preprocessing the MaxSAT formulas, Coprocessor 2 [77] was used with the Boolean constraint propagation, bounded variable elimination, failed literal elimination, probing, self-subsuming resolution, and blocked clause elimination techniques (configuration "[vpusb]+"). Experiments were performed on the set of 2014 MaxSAT Evaluation weighted partial instances with a time limit of 30 minutes (1800 seconds). As before, the experiments were run on machines with 32 GB of



(a) (b)

Figure 18: Evaluating assumption variable reuse on (a) crafted and (b) industrial benchmark instances.

memory and two Intel Xeon E5540 CPUs. We compare LMHS without preprocessing (LMHS) to LMHS with preprocessing (LMHS+pre) and LMHS with preprocessing when reusing assumption variables (LMHS+pre+reuse).

Figure 18 splits the results of these experiments by crafted (left) and industrial (right) benchmark instances. We note that SAT-based preprocessing alone does not result in a larger number of instances solved. In fact, its effect is clearly detrimental on the crafted benchmark instances. For both crafted and industrial instances, however, also reusing the variables created for preprocessing sees a significant improvement over only using preprocessing. For crafted instances this serves mostly to negate the detrimental effect of preprocessing. For industrial instances, we see an overall improvement in the number of instances solved.

For this work, we have only looked at reusing auxiliary variables created by preprocessing. Other questions relating to SAT-based preprocessing for MaxSAT remain. It could be worthwhile to identify problem domains which benefit from preprocessing as well as investigating different combinations of preprocessing techniques with MaxSAT solvers. Also of interest is the application of preprocessing techniques during solving (inprocessing) for other MaxSAT algorithms.

5.6 Incremental solving

In this section we consider incrementally solving MaxSAT problems. The MaxHS algorithm can be extended to allow for new constraints to be added

to the MaxSAT instance after it has been solved. Our implementation of MaxHS, the LMHS solver, includes this functionality and provides an external API for its use. Incremental MaxSAT has practical uses in for example model counting [52], as well as incremental problem refinement, as detailed in Section 6.

5.6.1 Model enumeration

Algorithm 16 enumerates all optimal solutions to a MaxSAT instance. The MaxHS algorithm is enclosed within the loop of Line 4. When the first solution is found, Line 12 records its cost as the optimal cost. On subsequent optimal solutions, Line 17 adds a single clause which forbids the current optimal model. The termination condition of Line 13 will be met when all optimal solutions (of which there are a finite amount) have been found.

Algorithm 16 Optimal model enumeration with MaxHS.

```
 1: function ENUMERATE(F_h, F_s, c)
 2:     K ← ∅
 3:     H ← ∅
 4:     while true do
 5:         while true do
 6:             (sat, κ, τ) ← SOLVESAT(F_h ∪ (F_s \ H))
 7:             if not sat then
 8:                 K ← K ∪ {κ}
 9:                 H ← SOLVEMCHS(K, c)
10:             else
11:                 break
12:         if opt is undefined then opt ← cost(τ)
13:         if cost(τ) > opt then
14:             break
15:         else
16:             yield τ
17:             F_h ← F_h ∪ {(⋁_{x_i=1}∈τ ¬x_i) ∨ (⋁_{x_i=0}∈τ x_i)}
```

The MaxHS algorithm also allows for a natural way of enumerating unique solutions in terms of satisfied soft clauses. This can be accomplished by simply replacing the refinement of $\mathcal{F}$ on Line 17 with adding the constraint $\sum_{C \in H} x_C - \sum_{C \in \mathcal{F}_s \setminus H} x_C < |H|$ to the hitting set IP (of Algorithm 4), followed by a re-computation of the hitting set. Likewise, we could enumerate the k best solutions by modifying the condition of Line 13 to only break after k solutions have been found, regardless of their optimality.

5.6.2 LMHS API

The uses of this type of incrementality are not limited to simply enumerating models. In fact, we can extend and constrain the working formula by adding arbitrary clauses between iterations. LMHS implements this functionality and provides an API for its use. An overview of this API follows.

- `reset` Resets the internal state of the MaxSAT solver, allowing a new instance to be started.
- `initialize` Initializes the MaxSAT solver and its components. Three variants of this method are offered. An instance can be initialized from a file, from clauses in memory, or as an empty instance to be built using the API.
- `getNewVariable` Requests a new variable from the internal SAT solver.
- `addHardClause` Adds a hard clause to the working MaxSAT instance.
- `addSoftClause` Adds a soft clause to the working MaxSAT instance. This automatically internally creates a blocking variable for the clause. This variable is returned by the function in case the user wishes to make use of it. As a rule, the blocking variable created will always have a larger index number than the last variable created with `getNewVariable`.
- `addCoreConstraint` If a subset of soft clauses is known to be unsatisfiable, it can be added as a core constraint using the blocking variables of the soft clause. This can speed up the solving process.
- `forbidLastModel` Internally creates a SAT constraint forbidding the previously found variable assignment.
- `forbidLastSolution` Internally creates an IP constraint forbidding the previously found set of satisfied soft clauses.
- `getOptimalSolution` Optimally solves the current MaxSAT instance.

LMHS provides C++ and C library interfaces to access these functions. In Section 6, we use this functionality to refine a formula between iterations of the outer loop of Algorithm 16 based previous found solutions.

6 Applying incremental MaxSAT

This section presents an application of incremental MaxSAT and the LMHS API to learning Bayesian network structure. Bayesian network structure learning [32] is an NP-complete [28] optimization problem. The problem

has a MaxSAT representation [32], but we take a different approach, using LMHS to incrementally solve subproblems within an existing integer linear programming approach. This work has been published in [100].

6.1 Learning optimal Bayesian network structures

Let $X = \{x_1, \ldots, x_n\}$ be a set of nodes representing random variables and let $\mathcal{P}_i = \{P_i : P_i \subseteq X \setminus \{x_i\}\}$. A candidate parent set of x_i is a set $P_i \in \mathcal{P}_i$. Given a directed graph $G = (X, E)$, the parent set of x_i is the set of nodes $\{x_j : (x_j, x_i) \in E\}$, i.e., the set of nodes from which there exists a directed edge to x_i. Conversely, a choice of parent sets $P_1, \ldots, P_n$ defines a graph with edges $\{(x_j, x_i) : x_j \in P_i\}$. If the resulting graph is acyclic (a DAG), the choice of sets P_i defines a Bayesian network (BN) structure. Finally, we have cost functions $s_i : \mathcal{P}_i \to \mathbb{R}^+$, which define a non-negative score for each candidate parent set of each x_i. The source of these scores is beyond the scope of this work, but in essence they describe some fitness function for the possible BNs with nodes X.

Given the preceding definitions, the problem of Bayesian network structure learning (BNSL) can be defined. The task in BNSL is to find a DAG $G^* = (X, E)$ that minimizes the sum of the costs of the parent sets of $X = \{x_1, \ldots, x_n\}$ defined by the graph, $P_1, \ldots, P_n$. More formally,

$$G^* \in \operatorname*{arg\,min}_{G \in \mathrm{DAGs}(X)} \sum_{i=1}^{n} s_i(P_i), \tag{8}$$

where $\mathrm{DAGs}(X)$ is the set of all DAGs with nodes X, and P_i is the parent set of x_i defined by G. In practice, the size $\mathcal{P}_i$ is often limited in some way as $X \setminus \{x_i\}$ has an exponential (in n) number of subsets.

We consider an IP formulation of BNSL based on the definition of BNSL given by Equation 8. Utilizing the concept of parent sets and their cost functions, we have

$$\text{minimize} \quad \sum_{x_i \in X} \sum_{S \in \mathcal{P}_i} s_i(S) \cdot P_i^S \tag{9}$$

$$\text{subject to} \quad \sum_{S \in \mathcal{P}_i} P_i^S = 1 \qquad \forall i \in 1 \ldots n, \tag{10}$$

$$\sum_{x_i \in X} \sum_{S \cap C = \emptyset} P_i^S \geq 1 \qquad \forall C \subset X, \tag{11}$$

$$P_i^S \in \{0, 1\} \qquad \forall i \in 1 \ldots n,\ S \in \mathcal{P}_i.$$

This formulation adds binary variables P_i^S which indicate that S is chosen as the parent set of i. Equation 10 specifies that each node must have exactly one parent set. The objective function is given in terms of these parent set variables as Equation 9. The acyclicity of the graph is encoded as the *cluster*

constraints of Equation 11. Given any subset of nodes, or *cluster*, C of X, these constraints state that at least one node $x_i \in C$ must have a parent set with no nodes in C. Equivalently, this states that no $C \subset X$ forms a cycle in the graph specified by the assignments to the parent set variables P_i^S.

6.2 GOBNILP

GOBNILP [13, 33] implements a search algorithm for learning optimal Bayesian networks within the SCIP integer programming framework [2]. Directly solving the BNSL IP of the previous section is not practical due to the exponential number of cluster constraints. Instead, GOBNILP utilizes a branch-and-cut algorithm which adds constraints as they are needed. An outline of the approach is given in Algorithm 17.

Algorithm 17 The branch-and-cut algorithm for a set of constraints c over binary variables and an objective function f.

```
    global x*                                          ▷ Best found integral solution.
 1: function BranchAndCut(c, f)
 2:     repeat
 3:         x ← SolveLPRelaxation(c, f)
 4:         if x* is defined and f(x) ≥ f(x*) then return x*
 5:         c_new ← FindCuttingPlanes(x)
 6:         c ← c ∪ c_new
 7:     until c_new = ∅
 8:     if x is integral then
 9:         x* ← x
10:         return x*
11:     else
12:         Choose a variable y for which x assigns a non-integral value.
13:         x_{y=0} ← BranchAndCut(c ∪ {y = 0}, f)
14:         x_{y=1} ← BranchAndCut(c ∪ {y = 1}, f)
15:         return arg min_{x̂ ∈ {x_{y=0}, x_{y=1}}} f(x̂)
```

The search starts on Line 3 by solving the linear programming (LP) relaxation of Equations 9–10 and any constraints that have already been added. The integrality constraint of the original IP, restricting variable values to $\{0, 1\}$, is relaxed to allow values in the range $[0, 1]$. Unlike the IP, this LP relaxation can be solved in polynomial time.

If the relaxation yields a result x worse than some known upper bound x^*, the search backtracks. Otherwise, x^* is updated and new cutting planes are computed on Line 5. These are linear inequalities which constrain the search space, and include any cluster constraints (of Equation 11) violated by x^*. If no cutting planes are found and x^* is integral (assigns integer values to each

variable), then x^* is an optimal solution for the current branch. Otherwise, the search branches on some variable y (Lines 12–14).

Algorithm 17 does not specify how the LP relaxation is solved or how cutting planes are found. GOBNILP solves the LP relaxation within SCIP using a standard LP solver such as SoPlex or CPLEX. The FindCutting-Planes function searches for standard IP cutting planes (from, e.g., Gomory or zero-half cuts), but also new cutting planes stemming from the cluster constraints of Equation 11 [13, 33]. If x^* violates the cluster constraint for a cluster C, a cutting plane

$$\sum_{x_i \in C} \sum_{S \cap C = \emptyset} P_i^S \geq 1 \tag{12}$$

can be added to refine the LP. On every call of FindCuttingPlanes, GOBNILP solves a new integer program, called a *sub-IP*, to find these cluster-based cutting planes.

The sub-IP is an integer program, the solutions of which are cyclic subgraphs (clusters) over the set of nodes X. For the remainder of this section, let $x^*(P_i^S) \in [0,1]$ be the value of P_i^S in the current best solution x^* for the LP relaxation. Note that, by construction, $\sum_{S \in \mathcal{P}_i} x^*(P_i^S) \leq 1$ holds for any solution x^* and node x_i. To construct the sub-IP, binary variables C_i are introduced for every x_i to indicate if x_i is in the found cluster C. Further variables J_i^S are introduced for each P_i^S for which $x^*(P_i^S) > 0$ and $S \neq \emptyset$. If $J_i^S = 1$, then x_i is in the cluster and one of its parents in S is also in the cluster. The sub-IP is then defined by Equations 13–16.

$$\text{maximize} \sum_{i=1}^{n} \sum_{S \in \mathcal{P}_i} x^*(P_i^S) \cdot J_i^S - \sum_{x_i \in X} C_i \tag{13}$$

$$\text{subject to } J_i^S \to C_i \qquad \forall J_i^S \tag{14}$$

$$J_i^S \to \bigvee_{x_j \in S} C_j \qquad \forall J_i^S \tag{15}$$

$$\sum_{i=1}^{n} C_i \geq 2 \tag{16}$$

$$C_i, J_i^S \in \{0,1\} \qquad \forall i \in 1 \ldots n,\ x^*(P_i^S) > 0$$

The objective function of Equation 13 serves two purposes. Its first part prioritizes parent sets with high values in x^* or alternatively nodes with many valid (wrt. Equation 14 and 15) parent sets. The second part, $-\sum_{x_i \in X} C_i$, gives preference to smaller clusters by applying a penalty proportional to cluster size. Equations 14 and 15 define semantics of the J_i^S variables. They enforce the conditions that if $J_i^S = 1$, then

- (Equation 14) $x_i \in C$, or equivalently $C_i = 1$, and

- (Equation 15) $S \cap C \neq \emptyset$, i.e. there exists $x_j \in S$ such that $C_j = 1$.

Equation 16 rules out trivial clusters (those containing less than 2 nodes).

As argued by Cussens in [33], any feasible solution to the sub-IP has cost greater than -1. Following Equation 12, they correspond to valid cutting planes. During search, GOBNILP generally generates multiple non-optimal feasible solutions before finding an optimal solution, and generates at least one cutting plane based on each of them.

6.3 Solving Sub-IPs with MaxSAT

We note that a simple conversion of the GOBNILP sub-IP of Equations 13–16 to MaxSAT exists, and use the same binary variables J_i^S and C_i for the reformulation. We then compare methods of enumerating the k best solutions to the sub-IP under different constraints, using our re-implementation of the MaxHS algorithm. Equation 14 can be represented as hard clauses $\neg J_i^S \vee C_i$, and Equation 15 as hard clauses $\neg J_i^S \vee \bigvee_{x_j \in S} C_j$. Equation 16 can be equivalently represented as hard clauses $\bigvee_{S \in \mathcal{P}_i} J_i^S$. This follows from Equations 14 and 15 (or their clausal equivalents). Together they imply that if $J_i^S = 1$ for any x_i and S, then both $C_i = 1$ and $C_j = 1$ for some $x_j \in S$, which fulfills the constraint of Equation 16.

Both parts of the sub-IP objective function of Equation 13 can be represented as sets of unit soft clauses. The first part, the sum of the weights of chosen parent sets J_i^S, is represented as unit soft clauses (J_i^S) with cost $x^*(P_i^S)$ for all x_i and $S \in \mathcal{P}_i$. The second part, which applies a unit penalty for each node in the cluster, is represented as unit clauses (C_i) with weight 1 for every x_i.

Combining these conversions, we have the following MaxSAT formulation of the sub-IP:

$$\begin{aligned} \mathcal{F}_h = &\Big(\bigwedge_i \bigwedge_{S \in \mathcal{P}_i} (\neg J_i^S \vee C_i)\Big) \wedge \\ &\Big(\bigwedge_i \bigwedge_{S \in \mathcal{P}_i} (\neg J_i^S \vee \bigvee_{x_j \in S} C_j)\Big) \wedge \bigwedge_i \Big(\bigvee_{S \in \mathcal{P}_i} J_i^S\Big) \\ \mathcal{F}_s = &\Big(\bigwedge_i \bigwedge_{S \in \mathcal{P}_i} (J_i^S, x^*(P_i^S))\Big) \wedge \bigwedge_i (C_i, 1). \end{aligned} \tag{17}$$

Given a solution τ to this MaxSAT instance, we have $\tau(C_i) = 1$ if and only if $x_i \in C$. A cluster C is defined by $\{C_i : \tau(C_i) = 1\}$. Next we consider different methods of adding constraints Equation 17 to rule out a found cluster C.

Ruling out only the found cluster To rule out exactly the found cluster C, the hard clause

$$\bigvee_{C_i:\tau(C_i)=1} \neg C_i \vee \bigvee_{C_i:\tau(C_i)=0} \neg C_i$$

is added to the MaxSAT instance. The constraint requires that either one of the selected nodes (C_i s.t. $\tau(C_i) = 1$) must be unselected, or one of the unselected nodes must be selected. Adding this constraint ensures that any subsequent solution will not correspond to the same cluster C.

Ruling supersets and subsets Given two clusters, C and C' such that $C \subset C'$, the cutting plane given by C can result in a more significant reduction of the LP search space than the cutting plane given by C' because the cutting plane constraint becomes more restrictive. Supersets of a cluster C can be ruled out by the hard clause

$$\bigvee_{C_i:\tau(C_i)=1} \neg C_i.$$

Adding this clause guarantees that C contains at least one node not present in any of the clusters given by the remaining MaxSAT solutions. Conversely, the clause

$$\bigvee_{C_i:\tau(C_i)=0} \neg C_i$$

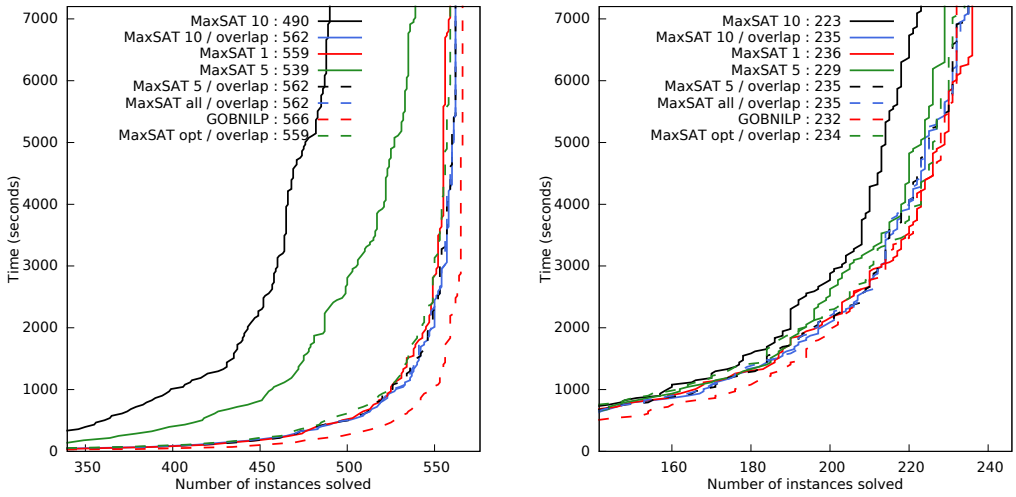
can be added to rule out subsets of C. This ensures that remaining solutions give clusters orthogonal to C in the sense that they must include some nodes not in C.

Ruling out overlapping clusters More orthogonal clusters—consisting of non-overlapping sets of nodes—yield cutting planes which prune entirely separate dimensions of the LP relaxation within GOBNILP. Such non-overlapping clusters can be found by adding hard clauses

$$\bigwedge_{C_i:\tau(C_i)=1} \neg C_i$$

to the MaxSAT instance after a model τ corresponding to a cluster C is found. These unit hard clauses ensure that no node in C is included in found clusters.

The incremental API of LMHS, detailed in Section 5.6, is used to add the appropriate type of constraint to the MaxSAT instance after every found solution. This is repeated until some desired condition is met, or until no more solutions exist. The next section discusses experiments with these constraints in more detail.



(a) (b)

Figure 19: An evaluation of MaxSAT-based cutting planes for (a) Bayesian network instances and (b) chordal Markov network instances.

6.4 Experiments

In this section, we replicate the experiments of [100] using an up-to-date version of the LMHS API. We use a set of 567 Bayesian network structure learning instances [76] and 285 chordal Markov network learning[11] instances [63]. The experiments were run on a cluster of 2.83GHz Intel Xeon E5440 machines with 32 GB of memory running Debian 8. A time limit of 7200 seconds (2h) was enforced.

Figure 19 shows a comparison of the various MaxSAT-based sub-IP methods to GOBNILP. The plots display the number of instances solved at any time within the time limit. Separate plots are given for chordal Markov network learning and Bayesian network learning instances. In the plots, the parameter after "MaxSAT" (1/5/10/all/opt) refers to the number of best sub-IP solutions solved for. This can be a fixed number, all solutions, or all optimal solutions. A further parameter "overlap" specifies that sub-IP solutions are incrementally refined to eliminate overlapping clusters.

The Bayesian network learning results of Figure 19 show the effectiveness of cluster refinement. While even finding a single optimal sub-IP solution performs well, some improvement is seen by finding multiple, non-overlapping clusters. We note that on these instances, the current version of LMHS performs slightly worse in comparison to GOBNILP than in the original experiments of [100]. This suggests that developments which improve performance on MaxSAT benchmarks in general do not necessarily translate

[11] Enabling the `gobnilp/noimmoralities` parameter in GOBNILP allows us to use an identical sub-IP model for learning chordal Markov networks

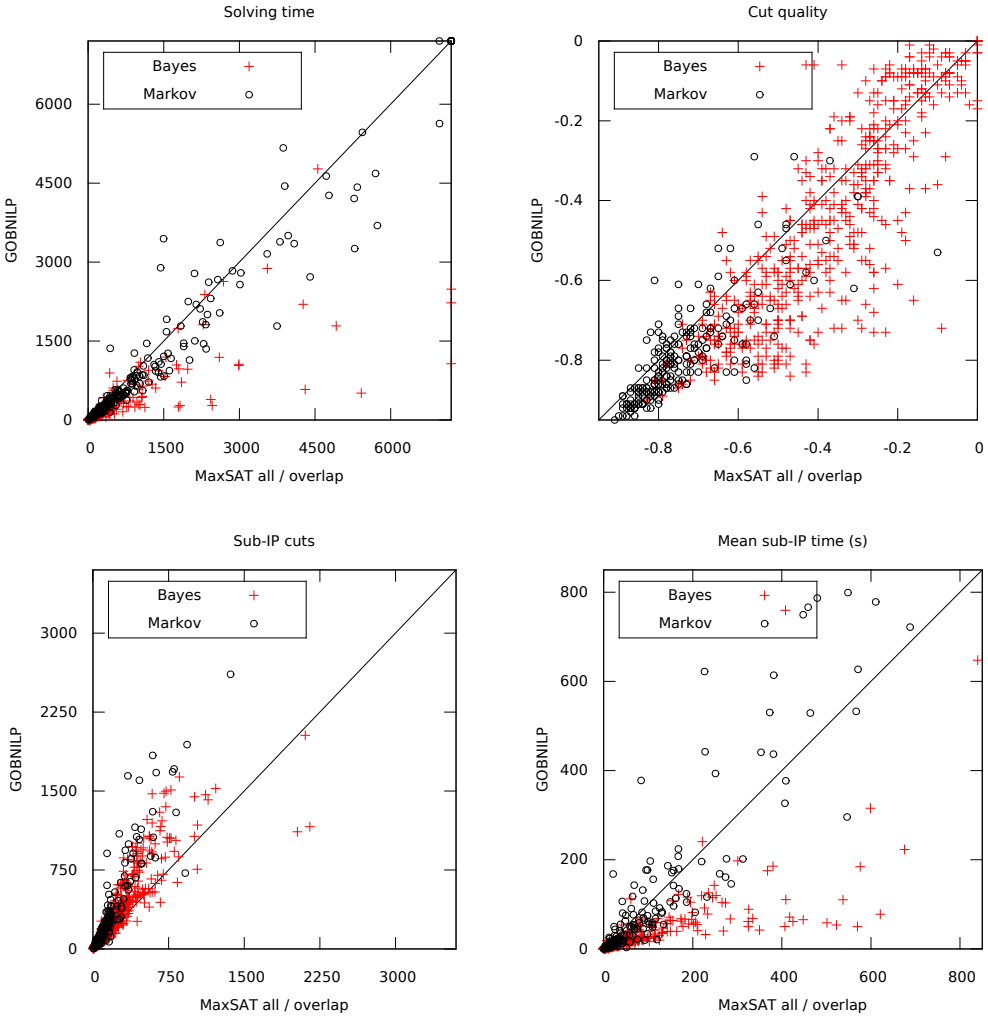


Figure 20: Comparisons of GOBNILP to "MaxSAT all / overlap".

to improvements on the comparatively small sub-IP instances, and that a sub-IP specific configuration could improve performance. On chordal Markov network instances, however, the results show less variance in performance, and the best MaxSAT-based methods slightly outperform GOBNILP.

Figure 20 provides a more detailed comparison between GOBNILP and MaxSAT-based sub-IP computations with non-overlapping clusters ("MaxSAT all / overlap"). We use different markers to distinguish between Bayesian network learning and chordal Markov network learning instances. The top left plot shows another comparison of solving times. Top right gives a comparison of cut quality (in terms of the sub-IP objective function). We see that the MaxSAT-based method adds higher quality cuts on average. A

comparison of mean sub-IP solving time per instance (bottom right) shows that although we need to add significantly fewer sub-IP cuts (bottom left), the sub-IPs take longer to solve on average. This comparison also serves to explain the difference in solving times between Bayesian network and Markov network instances. The MaxSAT solver seems to more efficiently solve sub-IPs of the chordal Markov network instances.

7 Conclusion

This thesis focused on analyzing and extending MaxHS, and examining applications arising from these extensions. We both reviewed our recent work and introduced new contributions. As part of this work we developed the LMHS MaxSAT solver, which won (among non-portfolio solvers) two categories at the 2015 MaxSAT Evaluation. Our other contributions include experiments with assumption shuffling to generate additional cores, a preliminary investigation of parallelizing the MaxHS algorithm, and a case study for applying external constraint propagators within a MaxSAT solver. We also review work on integrating SAT-based preprocessing [20, 21] into LMHS and gave an overview of its incremental MaxSAT API. An application of this API was show in previous work relating to incrementally constraining MaxSAT problems [100].

We see many promising areas of further study relating to this work. A more in-depth investigation of partitioning strategies for parallelization could further performance gains, as partitioning with SAT-based MaxSAT solvers has already seen some success [86, 93]. Our initial results with external constraint propagation show that it has the potential to be effective for individual problem domains. It is also worth investigating these external propagation techniques in conjunction with other MaxSAT algorithms. The assumption shuffling techniques investigated could find use in MUS extraction as well as core extraction in other MaxSAT algorithms. Finally, hitting-set based methods in general could be applied to other optimization paradigms such as ASP, as well as problems in knowledge representation and AI going beyond NP.

Acknowledgements

This work has been financially supported by Academy of Finland under grants 284591 and 251170 COIN Centre of Excellence in Computational Inference Research. I thank Matti Järvisalo and Petri Myllymäki for agreeing to act as the supervisors of this thesis, Jeremias Berg for collaboration on [21] and [20] as well as providing BTW-BNSL benchmark instances, Brandon Malone for collaboration on [100], and Matti Järvisalo for advice and guidance on this work.

References

[1] A. Abramé and D. Habet. Efficient application of Max-SAT resolution on inconsistent subsets. In *Proc. CP*, volume 8656 of *LNCS*, pages 92–107, 2014.

[2] T. Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[3] C. Ansótegui, M. Bonet, and J. Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013.

[4] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In *Proc. SAT*, volume 5584 of *LNCS*, pages 427–440. Springer, 2009.

[5] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In *Proc. SAT*, volume 5584 of *LNCS*, pages 167–180. Springer, 2009.

[6] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[7] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[8] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proc. IJCAI*, pages 399–404. AAAI Press, 2009.

[9] F. Azadivar and J. Wang. Facility layout optimization using simulation and genetic algorithms. *International Journal of Production Research*, 38(17):4369–4383, 2000.

[10] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proc. CP*, volume 2833 of *LNCS*, pages 108–122. Springer, 2003.

[11] V. Balabanov and A. Ivrii. Speeding up MUS extraction with preprocessing and chunking. In *Proc. SAT*, volume 9340 of *LNCS*, pages 17–32. Springer, 2015.

[12] J. F. Bard, G. Yu, and M. F. Arguello. Optimizing aircraft routings in response to groundings and delays. *IIE Transactions*, 33(10):931–947, 2001.

[13] M. Bartlett and J. Cussens. Advances in Bayesian network learning using integer programming. In *Proc. UAI*, pages 182–191. AUAI Press, 2013.

[14] A. Belov, M. Heule, and J. P. Marques-Silva. MUS extraction using clausal proofs. In *Proc. SAT*, volume 8561 of *LNCS*, pages 48–57. Springer, 2014.

[15] A. Belov, M. Järvisalo, and J. P. Marques-Silva. Formula preprocessing in MUS extraction. In *Proc. TACAS*, volume 7795 of *LNCS*, pages 108–123. Springer, 2013.

[16] A. Belov and J. P. Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Proc. FMCAD*, pages 37–40. IEEE, 2011.

[17] A. Belov, A. Morgado, and J. P. Marques-Silva. SAT-based preprocessing for MaxSAT. In *Proc. LPAR*, volume 8312 of *LNCS*, pages 96–111. Springer, 2013.

[18] J. Berg and M. Järvisalo. Optimal correlation clustering via MaxSAT. In *Proc. 2013 IEEE ICDM Workshops*, pages 750–757. IEEE Press, 2013.

[19] J. Berg, M. Järvisalo, and B. Malone. Learning optimal bounded treewidth bayesian networks via maximum satisfiability. In *Proc. AISTATS*, pages 86–95. JMLR, 2014.

[20] J. Berg, P. Saikko, and M. Järvisalo. Improving the effectiveness of SAT-based preprocessing for MaxSAT. In *Proc. IJCAI*, pages 239–245. AAAI Press, 2015.

[21] J. Berg, P. Saikko, and M. Järvisalo. Re-using auxiliary variables for MaxSAT preprocessing. In *Proc. ICTAI*. IEEE, 2015. To appear.

[22] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[23] A. Biere. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In *Pragmatics of SAT workshop at FLoC*, page 88. EasyChair, 2014.

[24] N. Bjørner and N. Narodytska. Maximum satisfiability using cores and correction sets. In *Proc. IJCAI*, pages 246–252. AAAI Press, 2015.

[25] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[26] K. Bunte, M. Järvisalo, J. Berg, P. Myllymäki, J. Peltonen, and S. Kaski. Optimal neighborhood preserving visualization by maximum satisfiability. In *Proc. AAAI*, pages 1694–1700. AAAI Press, 2014.

[27] K. Chandrasekaran, R. M. Karp, E. Moreno-Centeno, and S. Vempala. Algorithms for implicit hitting set problems. In *Proc. SODA*, pages 614–629. SIAM, 2011.

[28] D. M. Chickering. Learning Bayesian networks is NP-complete. In *Learning from Data*, pages 121–130. Springer, 1996.

[29] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[30] M. Codish and M. Zazon-Ivry. Pairwise cardinality networks. In *Proc. LPAR-16*, volume 6355 of *LNCS*, pages 154–172. Springer, 2010.

[31] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. STOC*, pages 151–158. ACM, 1971.

[32] J. Cussens. Bayesian network learning by compiling to weighted MAX-SAT. In *Proc. UAI*, pages 105–112. AUAI Press, 2008.

[33] J. Cussens. Bayesian network learning with cutting planes. In *Proc. UAI*, pages 153–160. AUAI Press, 2011.

[34] A. Darwiche. Tractable knowledge representation formalisms. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 2014.

[35] S. Daskalaki, T. Birbas, and E. Housos. An integer programming formulation for a case study in university timetabling. *European Journal of Operational Research*, 153(1):117–135, 2004.

[36] J. Davies. *Solving MAXSAT by Decoupling Optimization and Satisfaction*. PhD thesis, University of Toronto, 2013.

[37] J. Davies and F. Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proc. CP*, volume 6876 of *LNCS*, pages 225–239. Springer, 2011.

[38] J. Davies and F. Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proc. SAT*, volume 7962 of *LNCS*, pages 166–181. Springer, 2013.

[39] J. Davies and F. Bacchus. Postponing optimization to speed up MAXSAT solving. In *Proc. CP*, volume 8124 of *LNCS*, pages 247–262. Springer, 2013.

[40] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[41] J. Dunagan and S. Vempala. A simple polynomial-time rescaling algorithm for solving linear programs. *Mathematical Programming*, 114(1):101–114, 2008.

[42] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[43] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[44] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[45] N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal of Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[46] G. Elidan and S. Gould. Learning bounded treewidth bayesian networks. In *Proc. NIPS*, pages 417–424, 2009.

[47] J. Franco and J. Martin. A history of satisfiability. In *Handbook of Satisfiability*, chapter 1, pages 3–55. IOS Press, 2009.

[48] Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *Proc. SAT*, volume 4121 of *LNCS*, pages 252–265. Springer, 2006.

[49] M. Gebser, T. Janhunen, and J. Rintanen. SAT modulo graphs: Acyclicity. In *Proc. JELIA*, volume 8761 of *LNAI*, pages 137–151. Springer, 2014.

[50] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012.

[51] M. Gebser, B. Kaufmann, and T. Schaub. Advanced conflict-driven disjunctive answer set solving. In *Proc. IJCAI*, pages 912–918. AAAI Press, 2013.

[52] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*, chapter 20, pages 633–650. IOS Press, 2009.

[53] J. Guerra and I. Lynce. Reasoning over biological networks using maximum satisfiability. In *Proc. CP*, volume 7514 of *LNCS*, pages 941–956. Springer, 2012.

[54] F. Heras, A. Morgado, and J. P. Marques-Silva. Core-guided binary search algorithms for maximum satisfiability. In *Proc. AAAI*, pages 36–41. AAAI Press, 2011.

[55] F. Heras, A. Morgado, and J. P. Marques-Silva. An empirical study of encodings for group MaxSAT. In *Proc. Canadian Conference on AI*, volume 7310 of *LNCS*, pages 85–96. Springer, 2012.

[56] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proc. IJCAI*, pages 2318–2323. AAAI Press, 2007.

[57] IBM ILOG. CPLEX optimizer 12.6.0, 2014. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[58] M. Janota, R. Grigore, and J. P. Marques-Silva. Counterexample guided abstraction refinement algorithm for propositional circumscription. In *Proc. JELIA*, volume 6341 of *LNAI*, pages 195–207. Springer, 2010.

[59] M. Janota and J. P. Marques-Silva. Abstraction-based algorithm for 2QBF. In *Proc. SAT*, volume 6695 of *LNCS*, pages 230–244. Springer, 2011.

[60] M. Järvisalo, D. L. Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–94, 2012.

[61] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proc. STOC*, pages 38–49. ACM, 1973.

[62] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proc. PLDI*, pages 437–446. ACM, 2011.

[63] K. Kangas, M. Koivisto, and T. Niinimäki. Learning chordal Markov networks by dynamic programming. In *Proc. NIPS*, pages 2357–2365, 2014.

[64] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

[65] R. M. Karp. Reducibility among combinatorial problems. In *Proc. Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, 1972.

[66] H. Katebi, K. A. Sakallah, and J. P. Marques-Silva. Empirical study of the anatomy of modern SAT solvers. In *Proc. SAT*, volume 6695 of *LNCS*, pages 343–356. Springer, 2011.

[67] H. A. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Wiley, 1992.

[68] J. A. Kelner and D. A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *Proc. STOC*, pages 51–60. ACM, 2006.

[69] A. Klose and A. Drexl. Facility location models for distribution system design. *European Journal of Operational Research*, 162(1):4–29, 2005.

[70] M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa. QMaxSAT: A partial Max-SAT solver. *Journal of Satisfiability, Boolean Modeling and Computation*, 8(1/2):95–100, 2012.

[71] T. Kropf. *Introduction to Formal Hardware Verification*. Springer Science & Business Media, 2013.

[72] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[73] H. C. Lau. On the complexity of manpower shift scheduling. *Computers & Operations Research*, 23(1):93–102, 1996.

[74] C. M. Li and F. Manyà. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability*, chapter 19, pages 613–631. IOS Press, 2009.

[75] C. M. Li, F. Manyà, and J. Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

[76] B. Malone, K. Kangas, M. Järvisalo, M. Koivisto, and P. Myllymäki. Predicting the hardness of learning Bayesian networks. In *Proc. AAAI*, pages 2460–2466. AAAI Press, 2014.

[77] N. Manthey. Coprocessor 2.0–a flexible CNF simplifier. In *Proc. SAT*, volume 7317 of *LNCS*, pages 436–441. Springer, 2012.

[78] J. P. Marques-Silva. Practical applications of Boolean satisfiability. In *Proc. WODES*, pages 74–80. IEEE, 2008.

[79] J. P. Marques-Silva, M. Janota, A. Ignatiev, and A. Morgado. Efficient model based diagnosis with maximum satisfiability. In *Proc. IJCAI*, pages 1966–1972. AAAI Press, 2015.

[80] J. P. Marques-Silva and I. Lynce. Towards robust CNF encodings of cardinality constraints. In *Proc. CP*, volume 4741 of *LNCS*, pages 483–497. Springer, 2007.

[81] J. P. Marques-Silva and I. Lynce. On improving MUS extraction algorithms. In *Proc. SAT*, volume 6685 of *LNCS*, pages 159–173. Springer, 2011.

[82] J. P. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, chapter 4, pages 131–153. IOS Press, 2009.

[83] J. P. Marques-Silva and J. Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, abs/0712.1097, 2007.

[84] J. P. Marques-Silva and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, pages 171–182. Springer, 2011.

[85] J. P. Marques-Silva and K. A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *Proc. ICTAI*, pages 467–469. IEEE, 1996.

[86] R. Martins, V. M. Manquinho, and I. Lynce. On partitioning for maximum satisfiability. In *Proc. ECAI*, volume 242 of *FAIA*, pages 913–914. IOS Press, 2012.

[87] R. Martins, V. M. Manquinho, and I. Lynce. Open-WBO: A modular MaxSAT solver,. In *Proc. SAT*, volume 8561 of *LNCS*, pages 438–445. Springer, 2014.

[88] E. Moreno-Centeno and R. M. Karp. The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Operations Research*, 61(2):453–468, 2013.

[89] A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. P. Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.

[90] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. EDAC*, pages 530–535. ACM, 2001.

[91] A. Nadel, V. Ryvchin, and O. Strichman. Efficient MUS extraction with resolution. In *Proc. FMCAD*, pages 197–200. IEEE, 2013.

[92] N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In *Proc. AAAI*, pages 2717–2723. AAAI, 2014.

[93] M. Neves, R. Martins, M. Janota, I. Lynce, and V. Manquinho. Exploiting resolution-based representations for MaxSAT solving. In *Proc. SAT*, volume 9340 of *LNCS*. Springer, 2015.

[94] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[95] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[96] O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[97] K. Pipatsrisawat and A. Darwiche. Clone: Solving weighted Max-SAT in a reduced search space. In *Proc. AI*, volume 4830 of *LNCS*, pages 223–233. Springer, 2007.

[98] S. Prestwich. CNF Encodings. In *Handbook of Satisfiability*, chapter 2, pages 75–97. IOS Press, 2009.

[99] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[100] P. Saikko, B. Malone, and M. Järvisalo. MaxSAT-based cutting planes for learning graphical models. In *Proc. CPAIOR*, volume 9075 of *LNCS*, pages 347–356. Springer, 2015.

[101] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.

[102] R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[103] V. D. Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[104] C. Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proc. CP*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005.

[105] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pages 466–483. Springer, 1983.

[106] C. Zhu, G. Weissenbacher, and S. Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In *Proc. FMCAD*, pages 63–66. IEEE, 2011.