Solving MAXSAT by Decoupling Optimization and Satisfaction

by

Jessica Davies

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Solving MAXSAT by Decoupling Optimization and Satisfaction

Jessica Davies

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2013

Many problems that arise in the real world are difficult to solve partly because they present computational challenges. Many of these challenging problems are optimization problems. In the real world we are generally interested not just in solutions but in the cost or benefit of these solutions according to different metrics. Hence, finding optimal solutions is often highly desirable and sometimes even necessary. The most effective computational approach for solving such problems is to first model them in a mathematical or logical language, and then solve them by applying a suitable algorithm.

This thesis is concerned with developing practical algorithms to solve optimization problems modelled in a particular logical language, MAXSAT. MAXSAT is a generalization of the famous Satisfiability (SAT) problem, that associates finite costs with falsifying various desired conditions where these conditions are expressed as propositional clauses. Optimization problems expressed in MAXSAT typically have two interacting components: the logical relationships between the variables expressed by the clauses, and the optimization component involving minimizing the falsified clauses. The interaction between these components greatly contributes to the difficulty of solving MAXSAT.

The main contribution of the thesis is a new hybrid approach, MAXHS, for solving MAXSAT. Our hybrid approach attempts to decouple these two components so that each can be solved with a different technology. In particular, we develop a hybrid solver that exploits two sophisticated technologies with divergent strengths: SAT for solving the logical component, and Integer Programming (IP) solvers for solving the optimization component. MAXHS automatically and incrementally splits the MAXSAT problem into two parts that are given to the SAT and IP solvers, which work together in a complementary way to find a MAXSAT solution. The thesis investigates several improvements to the MAXHS approach and provides empirical analysis of its behaviour in practise. The result is a new solver, MAXHS, that is shown to be the most robust existing solver for MAXSAT.

Acknowledgements

To all of the people that I got to know during my graduate studies, thank you for the different ways you helped me along the way. Thank you to the members of my Committee, Sheila McIlraith and Toniann Pitassi, for your advice and encouragement. I was very fortunate to be advised by Fahiem Bacchus. I will always appreciate his insight and patience, which guided and inspired me along this path. Finally, thank you to my parents for your constant love and support. I dedicate this thesis to you both.

Preface

The research I conducted during the course of my PhD studies involved projects in several different areas of computer science, including automated reasoning, computational social choice, and formal methods. This thesis is concerned with a particular topic that became the focus of my research: practical algorithms for solving MAXSAT. Most of the results in the thesis have been published in conference papers, Chapter 3 in (Davies and Bacchus, 2011), Chapter 4 in (Davies and Bacchus, 2011) and (Davies and Bacchus, 2013) and Chapter 6 in (Davies, Cho, and Bacchus, 2010).

I was a co-author of the following papers during my PhD.

- Journal Papers

  1. Davies, J., Katsirelos, G., Narodystka, N., Walsh, T. and Xia, L. Complexity of and Algorithms for the Manipulation of Borda, Nanson and Baldwin's Voting Rules. *Artificial Intelligence*. Accepted, 2012.

  2. Simmonds, J., Davies, J., Gurfinkel, A., and Chechik, M. Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC. *International Journal on Software Tools for Technology Transfer (STTT)*. 12(5), p.319-335, 2010.

- Conference Papers

  1. Davies, J., and Bacchus, F. Exploiting the Power of MIP Solvers in MAXSAT. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, 2013.

  2. Davies, J., Narodytska, N., and Walsh, T. Eliminating the Weakest Link: Making Manipulation Intractable? In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2012.

3. Davies, J., Katsirelos, G., Narodystka, N., and Walsh, T. Complexity of and Algorithms for Borda Manipulation. **AAAI outstanding paper award**. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2011.

4. Davies, J., and Bacchus, F. Solving MAXSAT by Solving a Sequence of Simpler SAT Instances. In *Principles and Practice of Constraint Programming (CP)*, 2011.

5. Davies, J., Cho, J., and Bacchus, F. Using Learnt Clauses in MAXSAT. In *Principles and Practice of Constraint Programming (CP)*, 2010.

6. Simmonds, J., Davies, J., Gurfinkel, A., and Chechik, M. Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, 2007.

7. Davies, J. and Bacchus, F. Using More Reasoning to Improve #SAT Solving. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2007.

8. Samulowitz, H., Davies, J., and Bacchus, F. Preprocessing QBF. In *Principles and Practice of Constraint Programming (CP)*, 2006.

- Workshop Papers

1. Davies, J., Katsirelos, G., Narodystka, N., and Walsh, T. An Empirical Study of Borda Manipulation. *Third International Workshop on Computational Social Choice (ComSoc 2010)*. Dusseldorf, Germany, 2010.

2. Davies, J. and Bacchus, F. Distributional Importance Sampling for Approximate Weighted Model Counting. *The First Workshop on Counting Problems in CSP and SAT, and Other Neighbouring Problems (Counting 2008)*. Sydney, Australia, 2008.

Contents

Chapter 1

Introduction

Many problems that arise in the real world are difficult to solve partly because they present computational challenges. Furthermore, it is often important to find not just any solution to the problem, but rather one that is "best" according to some objective. In this case, the problem falls into the class of optimization problems. The most effective approach to solving such problems is to first model them in a mathematical or logical language, and then solve them by applying a suitable algorithm. This thesis is concerned with developing practical algorithms to solve optimization problems modelled in a particular logical language, MAXSAT. MAXSAT is a generalization of the famous Satisfiability problem.

Both SAT and MAXSAT deal with propositional logic formulas in Conjunctive Normal Form.[1] In the SAT problem, the goal is to find a truth assignment that satisfies all of the clauses if one exists, and to report that no satisfying assignment exists otherwise. However, when there is no satisfying assignment, it may still be useful to find the truth assignment that satisfies as many clauses as possible, and this is the goal of MAXSAT. A solution to a MAXSAT instance is a truth assignment that satisfies the maximal number of clauses.

[1] Any Boolean function can be represented by a formula in CNF. A formula is in CNF if it is a conjunction of clauses, each of which is a disjunction of literals (Boolean variables or their negations).

In order to facilitate the modelling of a variety of optimization problems, where some constraints may be more important to satisfy than others, the MAXSAT problem can be extended by associating different costs with falsifying different clauses. In this case, it is natural to cast MAXSAT in terms of *minimizing* the total cost of the falsified clauses.[2] This thesis addresses the most general form of MAXSAT instances, commonly referred to as Weighted Partial MAXSAT.[3] Each propositional clause is associated with a weight that is either a positive integer or infinity. The solution to the MAXSAT instance is a truth assignment to the propositional variables that minimizes the total weight of falsified clauses, called the cost of the assignment. Hence infinite weight clauses express conditions that must be satisfied.

To illustrate how MAXSAT is used to model optimization problems, we describe a natural encoding of the well-known Traveling Salesman Problem (TSP). In TSP, we imagine that there is a salesman who needs to visit a number of cities (each at most once except for the starting city), ending up back at the first city. Such an itinerary is called a tour. For the sake of simplicity, we assume that it is possible to travel directly between any of the cities, and that the distance in both directions is equal (this is the symmetric TSP on a complete graph). A solution to the TSP is a tour of minimal length, where the length is calculated as the sum of the distances.

In order to model TSP as MAXSAT, let n be the number of cities. We introduce $n(n+1)$ Boolean variables x_{ij} to represent visiting city i at time step j, where $1 \leq i \leq n$ and $1 \leq j \leq n+1$. We add clauses over these variables to enforce the tour constraints, as follows. In order to specify that some city is visited at every time step, we introduce $n+1$ clauses $\{x_{1j}, x_{2j}, ..., x_{nj}\}$, $1 \leq j \leq n+1$. These clauses state that at time j, the salesman is visiting at least one of the cities. We must also specify that it is impossible to be in more than one city at the same time, using the clauses $\{\neg x_{ij}, \neg x_{kj}\}$ for all pairs

[2] Our definition of MAXSAT as a minimization problem is equivalent to the familiar definition because minimizing the number of falsified clauses necessarily maximizes the number of satisfied clauses.

[3] In the remainder of the thesis whenever we refer to MAXSAT without any qualifications we mean Weighted Partial MAXSAT.

of distinct cities $i \neq k$, $1 \leq i, k \leq n$ and all time steps $1 \leq j \leq n+1$. This ensures that if some x_{ij} is *true* all other x_{kj} for $k \neq i$ must be *false*, i.e., if the salesman is visiting city i at time step j they cannot also be visiting another city k at the same time. Now we must express the property that every city be visited. The clauses $\{x_{i1}, ...x_{in}\}$ for each city i capture this property since they require city i to be visited at one of the first n time steps (at time step $n+1$ the salesman revisits the starting city). Finally, we add the two clauses $\{\neg x_{i1}, x_{in+1}\}$ and $\{x_{i1}, \neg x_{in+1}\}$ for every city i which ensure that the first and last city are the same. We associate an infinite weight with each of the clauses mentioned so far, to make sure that the MAXSAT solution will satisfy all of these constraints.

Now, any assignment to the variables that satisfies the above mentioned clauses specifies a legal tour of the cities. It remains to express the different distances between the cities, which is achieved by using additional clauses with finite weights. For every pair of cities $i \neq k$, we add n clauses $\{\neg x_{ij}, \neg x_{kj+1}\}$ for $1 \leq j \leq n$, each with weight equal to the distance between cities i and k. A truth assignment falsifies such a clause if and only if it assigns *true* to both x_{ij} and x_{kj+1}. This represents moving from city i at time step j to city k at time step $j+1$. If a truth assignment falsifies such a clause, a cost equal to the distance between i and k will be incurred. Thus, any truth assignment that satisfies all of the clauses with infinite weight from above, and therefore corresponds to a valid tour, will have cost equal to the length of the tour. Therefore the MAXSAT solution, which finds a minimal cost truth assignment, corresponds to a minimal length tour as required to solve the TSP.

This thesis is concerned with practical, exact algorithms to solve the MAXSAT problem. The MAXSAT problem is complete for the complexity class $\mathrm{FP}^{\mathrm{NP}}$, which contains many important discrete optimization problems including TSP. Therefore, if we could develop a robust and efficient MAXSAT solver, it could be used to solve many different kinds of optimization problems by translating them to MAXSAT. However, since MAXSAT is $\mathrm{FP}^{\mathrm{NP}}$-complete, it is an even harder problem than SAT. So developing a robust and efficient

MAXSAT solver is a very challenging goal, and ultimately any MAXSAT algorithm will have limitations.

Yet we are encouraged by the example of SAT solvers, which although also tackling hard problems (in this case NP-complete), often perform very effectively as a black-box solver for many real-world applications. Since MAXSAT is a natural extension of SAT we believe that in the future, MAXSAT solvers will also see a rise in popularity, based on providing a convenient and efficient approach for solving optimization problems.

Many MAXSAT solvers have been developed since the mid 1990's, and they have been successfully applied to solve a variety of problems. For example, the Electronic Design Automation (EDA) domain is a natural application area for MAXSAT since the problems deal with many logical constraints that are easily expressed as conjunctions of clauses. EDA tasks such as circuit routing and design debugging can be encoded as MAXSAT and efficiently solved using MAXSAT solvers that exploit SAT technology. Furthermore, MAXSAT is used to model and solve problems in an increasing diversity of applications, such as planning, scheduling, probabilistic inference, and bioinformatics as well.

The underlying algorithms implemented by existing state-of-the-art MAXSAT solvers fall into two main classes: Branch and Bound search, and solving a sequence of decision problems. In the latter class, most MAXSAT solvers exploit existing SAT technology, by converting the decision problems to CNF and passing them to a SAT solver. In contrast, the Branch and Bound solvers perform a backtracking search through the space of truth assignments, and rely on calculating bounds to prune the search tree. It has been observed that solvers that follow these two approaches tend to work efficiently on different types of MAXSAT instances. Very generally, Branch and Bound solvers do better on "crafted" instances, e.g. graph optimization problems like TSP, while MAXSAT solvers that use a sequence of SAT problems approach are able to tackle the very large instances arising in "industrial" applications like EDA. It is possible to speculate as to why this is the case, but no experimentally or theoretically verified explanation for the differing

behaviour of MAXSAT solvers currently exists.

As we described above, MAXSAT is a natural language to represent optimization problems and already many MAXSAT solvers have been developed. So it is striking that in practice, the most commonly applied optimization technology is not MAXSAT but Integer Programming (IP). For example, the current best solver for TSP is based on the techniques commonly used to solve IP problems (Applegate, Bixby, Chvátal, and Cook, 2007). IP is a formalism for representing linear optimization problems over discrete valued variables, and it is well studied in the Operations Research community. In IP, the goal is to find an assignment that satisfies a set of feasibility constraints, which are arbitrary linear inequalities over integer variables, and that minimizes or maximizes a linear objective function. In the Operations Research community, sophisticated algorithms to solve IP models have been developed. The main approach is Branch and Cut, which relies on solving linear relaxations of the IP.[4] The best existing implementations of Branch and Cut are those incorporated in proprietary Mixed Integer Programming (MIP) packages, such as IBM ILOG's CPLEX. These packages are an important tool in many industries, e.g. airline crew scheduling.

Given the proven popularity of MIP solvers, it is natural to ask whether they are also effective on MAXSAT instances. There is a simple encoding of MAXSAT to IP and we show in Section 3.7 that the MIP solver CPLEX actually solves many MAXSAT instances faster than existing specialized MAXSAT solvers. However, it also fails on many other instances that specialized MAXSAT solvers can solve.

Although in principle SAT can also be translated to IP and solved using MIP solvers, the performance of MIP solvers on SAT instances is known to be very poor. This is because the special-purpose techniques employed by SAT solvers are particularly suited to handling a large number of logical constraints, which occur in MAXSAT instances as well. Thus it is not surprising that MAXSAT solvers that exploit SAT technology are

[4]The linear relaxation of an IP is an LP (Linear Program) that is derived by lifting the restriction that the variables take on integer values.

able to solve many MAXSAT instances that are too challenging for MIP solvers. Hence, a promising approach to create a more robust MAXSAT solver is to investigate ways to combine SAT and MIP techniques.

In this thesis we introduce a new hybrid approach for solving MAXSAT, that exploits two sophisticated technologies with divergent but complementary strengths: SAT and Integer Programming solvers. The proposed MAXHS approach decouples the satisfaction and optimization aspects of MAXSAT. Specifically, MAXHS automatically and incrementally splits the MAXSAT problem into two parts, given to the SAT and MIP solvers respectively, and facilitates communication between the two sub-solvers so that they work together to solve the MAXSAT problem. The SAT and MIP solvers are used as black-boxes. Thus MAXHS is able to exploit the different abilities of SAT and MIP solvers to effectively solve the MAXSAT problem, automatically and without touching the SAT and MIP solvers' internal algorithms.

We demonstrate through an extensive empirical evaluation that our new MAXSAT solver based on the MAXHS approach is more robust than any existing MAXSAT solver. Furthermore, the hybrid MAXHS solver performs significantly better than using a MIP solver alone to solve MAXSAT.

The remainder of the thesis is organized as follows. In Chapter 2, the MAXSAT problem is defined, applications of MAXSAT are listed, and the main existing approaches for solving MAXSAT are explained in detail. The MAXHS approach is introduced in Chapter 3, and the implementation of the MAXHS solver is also described. Section 3.7 describes the experimental setup used throughout the thesis, and is followed by an empirical comparison of the initial version of the MAXHS solver and existing state-of-the-art solvers. The following two chapters (Chapters 4 and 5) consider two significant enhancements to the basic MAXHS approach, based on increasing the information provided to the MIP solver, that result in greater robustness. Chapter 4 proposes to generalize the type of constraints MAXHS gives to the optimization subproblem, while Chapter 5 studies how

maxhs can utilize cheaper approximations of the optimization subproblem. The thesis changes direction in Chapter 6, where some of the maxhs ideas are applied within a Branch and Bound algorithm for maxsat. Finally, Chapter 7 concludes the thesis with a summary of the contribution and ideas for future work.

Chapter 2

Background

2.1 Introduction

The Maximum Satisfiability (MAXSAT) problem is an extension of Satisfiability (SAT) that is able to represent optimization problems. Many kinds of real-world optimization problems can be expressed as MAXSAT, and then solved using a general purpose solver. This approach is often a very efficient method of getting an optimal answer to the original problem.

MAXSAT, like SAT, deals with formulas in Conjunctive Normal Form, which consist of a conjunction of clauses, where each clause is a disjunction of propositional literals. In the SAT problem, the goal is to find a truth assignment that satisfies all clauses of the formula, and to report that the formula is unsatisfiable if such an assignment does not exist. However, if the formula is unsatisfiable, it may still be useful to find a truth assignment that satisfies as many of the clauses as possible, and finding such an assignment is the aim of MAXSAT. If some of the clauses are more important to satisfy than others, this can be enforced by associating a cost with falsifying each clause. Furthermore, if some clauses are mandatory to satisfy, they can be given infinite cost. Section 2.2 defines the MAXSAT problem and highlights theoretical results that suggest

it is an important, but hard, problem to solve.

Solving MAXSAT is also important from a practical perspective. Many interesting real-world problems have been expressed as MAXSAT, from areas such as electronic design automation (EDA), planning, probabilistic inference, and software upgradeability. There are potential applications in bioinformatics, scheduling, and combinatorial auctions as well. Given the success story of SAT solvers, which can be treated as a blackbox to solve very large problems arising from industrial applications, and the close relationship of MAXSAT and SAT, it is possible that future MAXSAT solvers will play a significant role in solving real-world optimization problems. Section 2.3 begins by showing how a familiar combinatorial optimization problem can be naturally expressed as MAXSAT, and then describes some real-world applications of MAXSAT in detail.

The existence of important real-world applications motivates the development of efficient MAXSAT solvers. The first MAXSAT solvers were based on the Branch and Bound algorithm, an established approach for solving combinatorial optimization problems. Branch and Bound performs a depth-first search over partial assignments, keeping track of the best complete assignment found so far. At each node of the search tree, a bound on the quality of any complete assignment below the current node is calculated. This bound can be used to prune the subtree. Typically, the performance of Branch and Bound is significantly affected by the strength of this bound. Section 2.5.1 explains the specialized bounds used by MAXSAT solvers.

Some MAXSAT instances, especially those arising in industrial applications like EDA, are too challenging for existing Branch and Bound MAXSAT solvers, yet can be easily refuted by a state-of-the-art SAT solver. Such instances can sometimes be solved by instead converting the MAXSAT problem to a sequence of SAT problems, and then exploiting a SAT solver as a blackbox to solve each one. These solvers work by adding fresh variables to the soft clauses, called relaxation variables since they can effectively remove soft clauses from the problem. Constraints over the relaxation variables are added to

control how many clauses are relaxed. The various methods used to reduce both the number of calls to the SAT solver and the size or difficulty of the SAT instances, are discussed in Section 2.5.2.

2.2 The MAXSAT Problem

We first define the MAXSAT problem, as it will be used throughout the thesis. An instance of the MAXSAT problem is given by a CNF formula and a weight for each clause in the formula.

Definition 1 (Instance of MAXSAT)**.** *An instance of the* MAXSAT *problem is given by a propositional formula in Conjunctive Normal Form, $\mathcal{F}$, and a weight $wt(c) \in \mathbb{N}^+ \cup \{\infty\}$ for every clause c in $\mathcal{F}$.*

Although Definition 1 specifies that the weights are integers, in practise the algorithms introduced in this thesis can handle finite precision real costs as easily as integers. The weights are assumed to be greater than zero, since clauses with weight zero can be removed from $\mathcal{F}$ without impact.

Clauses might be hard clauses, indicated by them having infinite weight. Clauses with finite weight are called soft clauses. $hard(\mathcal{F})$ is used to indicate the hard clauses of $\mathcal{F}$ and $soft(\mathcal{F})$ the soft clauses. In practise, the infinite weight is represented by a suitable large integer, for example, a value larger than the sum of the weights of all soft clauses.

Definition 2 (Hard and Soft Clauses)**.** $hard(\mathcal{F}) = \{c \in \mathcal{F} : wt(c) = \infty\}$ *is the set of hard clauses of* MAXSAT *instance $\mathcal{F}$.* $soft(\mathcal{F}) = \{c \in \mathcal{F} : wt(c) \neq \infty\}$ *is the set of soft clauses of $\mathcal{F}$. It is the case that $\mathcal{F} = hard(\mathcal{F}) \cup soft(\mathcal{F})$ and $hard(\mathcal{F}) \cap soft(\mathcal{F}) = \emptyset$.*

In the literature, the type of MAXSAT instance defined in Definition 1 is often called Weighted Partial MAXSAT. Three common restrictions include Partial MAXSAT, where all soft clauses have weight 1, and Weighted MAXSAT, where there are no hard clauses.

The "unweighted" MAXSAT problem is to satisfy as many clauses as possible; there are no hard clauses, and all clause weights are 1.

The weight $wt(c)$ associated with a clause c of a MAXSAT instance $\mathcal{F}$ represents the cost of falsifying that clause. The goal of solving MAXSAT is to find a truth assignment that falsifies the least weight of clauses, i.e. that incurs the least cost. The cost of a truth assignment is defined as follows.

Definition 3 (Cost of a Truth Assignment)**.** *If π is a truth assignment to the variables in* MAXSAT *instance $\mathcal{F}$ then $cost(\pi, \mathcal{F})$ is the sum of the weights of the clauses falsified by π: $cost(\pi, \mathcal{F}) = \sum_{\{c \in \mathcal{F} \mid \pi \not\models c\}} wt(c)$.*

Sometimes it will also be necessary to refer to the cost of a set of clauses.

Definition 4 (Cost of a Clause Set)**.** *If H is a set of weighted clauses then $cost(H)$ is the sum of the clause weights in H: $cost(H) = \sum_{c \in H} wt(c)$.*

To solve a MAXSAT instance, a truth assignment of minimal cost must be found.

Definition 5 (MAXSAT Solution)**.** *If $\mathcal{F}$ is a* MAXSAT *instance then a solution to $\mathcal{F}$ is a truth assignment π to the variables of $\mathcal{F}$ such that $cost(\pi, \mathcal{F})$ is minimal.*

We use $mincost(\mathcal{F})$ to denote the cost of a solution to MAXSAT instance $\mathcal{F}$.

Definition 6 ($mincost(\mathcal{F})$)**.** *$mincost(\mathcal{F})$ denotes the cost of a solution to* MAXSAT *instance $\mathcal{F}$.*

Solving MAXSAT can be equivalently defined in terms of maximizing the sum of the weights of the *satisfied* clauses. In this thesis, the MAXSAT problem is defined in terms of minimization of cost because it allows hard clauses to be treated the same as other weighted clauses.

If the hard clauses of MAXSAT instance $\mathcal{F}$ are not satisfiable, then every truth assignment will falsify at least one hard clause and therefore $mincost(\mathcal{F}) = \infty$.

Definition 7 (Unsatisfiable). *If* $mincost(\mathcal{F}) = \infty$ *then the* MAXSAT *instance* $\mathcal{F}$ *is said to be unsatisfiable.*

In the remainder of the thesis, it is assumed that $hard(\mathcal{F})$ is satisfiable and that $\mathcal{F} = hard(\mathcal{F}) \cup soft(\mathcal{F})$ is unsatisfiable. It is straightforward to extend all of the results to deal with these corner cases. Furthermore, from a practical point of view both conditions can be easily tested with a SAT solver and if either is violated the MAXSAT solution is immediately known: if $hard(\mathcal{F})$ is unsatisfiable then $mincost(\mathcal{F}) = \infty$ and any truth assignment is a solution; and if $\mathcal{F}$ is satisfiable then $mincost(\mathcal{F}) = 0$ and the SAT solution is also a MAXSAT solution.

We will also need two additional basic definitions. In the context of SAT, an UNSAT core is any unsatisfiable subset of the formula. In MAXSAT, where some clauses are hard and some are soft, we define a core similarly as a subset of the *soft* clauses that can not be satisfied at the same time as the hard clauses.

Definition 8 (Core). *A* ***core*** κ *for a* MAXSAT *formula* $\mathcal{F}$ *is a subset of* $soft(\mathcal{F})$ *such that* $\kappa \cup hard(\mathcal{F})$ *is unsatisfiable.*

Note that every truth assignment falsifies at least one clause of $\kappa \cup hard(\mathcal{F})$, and any truth assignment that satisfies the hard clauses will falsify at least one clause in κ itself.

Finally, the decision version of the MAXSAT problem is defined as follows.

Definition 9 (MAXSAT Decision Problem). *The* MAXSAT *decision problem is to determine for a particular* $k \in \mathbb{R}^+ \cup \infty$ *if there is a truth assignment* π *to the variables of* MAXSAT *instance* $\mathcal{F}$ *such that* $cost(\pi, \mathcal{F}) = k$.

The MAXSAT decision problem is NP-complete and MAXSAT is NP-hard (Cook, 1971), so it is not expected that an algorithm to efficiently solve all instances will be found.

Another way to understand the difficulty of solving MAXSAT is to consider the complexity of solving MAXSAT given access to a SAT oracle.

Definition 10 ($\mathrm{FP}^{\mathrm{NP}}$). *$FP^{NP}$ is the class of all functions from strings to strings that can be computed in polynomial time by a deterministic Turing machine with a SAT oracle.*

Theorem 1. *(Papadimitriou, 1994) The* MAXSAT *problem with finite integer weights is complete for the complexity class FP^{NP}.*

Many other well-known optimization problems are also $\mathrm{FP}^{\mathrm{NP}}$-complete, including The Traveling Salesman Problem, Knapsack, weighted Max-Cut and weighted Bisection Width.[1]

2.3 Applications of MAXSAT

There are many potential applications of MAXSAT. For example, wherever SAT is currently employed to solve a real-world problem, if there are preferences over the SAT solutions they can be captured using MAXSAT, in order to find a preferred solution. Or in the case that the SAT encoding of a problem is over-constrained and there is no satisfying assignment, the MAXSAT solution that satisfies as many constraints as possible may still be of practical utility. Going farther, any optimization problem in $\mathrm{FP}^{\mathrm{NP}}$, which also includes many practical optimization problems, can be expressed as MAXSAT. Of course, MAXSAT is more suited to some applications than others, since for example, there may be no reasonably compact MAXSAT model. There are alternative ways to model and solve optimization problems that generalize MAXSAT, such as Weighted Constraint Satisfaction and Integer Linear Programming. In this section we first explain how a familiar

[1] An instance of the Traveling Salesman Problem is given by a list of n cities and a nonnegative integer distance between each pair of cities. The problem is to find a shortest tour, that begins and ends at the same city and visits each other city exactly once. An instance of the Knapsack problem is given by a set of n items, each with a positive integer value v_i and weight w_i, and a maximum total weight G. The problem is to find a subset of the items such that their total weight is at most G and their total value is maximized. The weighted Max-Cut problem is posed on an undirected graph with positive edge weights, and its solution is a partition of the vertices into two sets such that the total weight of the edges that cross the partition is maximized. The weighted Bisection Width problem is similar to weighted Max-Cut except that the desired partition must divide the vertices into two *equally* sized sets such that the total weight of the edges crossing the bisection is *minimized.*

optimization problem can be easily encoded as MAXSAT. Then we go on to highlight real-world problems where MAXSAT solvers have been applied successfully.

2.3.1 Max-Cut

The Maximum Cut problem is to find a partition of the vertices of an undirected graph into two sets such that the number of edges going between the sets is maximized. The Weighted Maximum Cut problem adds positive weights to the edges of the graph, and the goal becomes to maximize the sum of the weights of the edges in the cut. Next we give two possible MAXSAT encodings of the Weighted Maximum Cut problem.

Binary Clause Encoding

Let $G = (V, E)$ be an undirected graph with edge weights given by the function $\omega : E \to \mathbb{R}^{\geq 0}$. For every vertex $i \in V$ in the graph, the MAXSAT instance has a boolean variable v_i that represents whether or not vertex i is in the first partition. For each edge (i, j) in the graph, there are two binary soft clauses in the MAXSAT instance of the form $(v_i \vee v_j)$ and $(\neg v_i \vee \neg v_j)$. The cost of each of these two clauses is equal to the weight of the edge. So $\mathcal{F} = \{(v_i \vee v_j), (\neg v_i \vee \neg v_j) : (i, j) \in E\}$ where $wt((\ell_i \vee \ell_j)) = \omega((i, j))$ for each $(\ell_i \vee \ell_j) \in \mathcal{F}$.

Therefore, every time two vertices i, j that are connected by an edge are assigned to the same partition, a cost of $\omega((i, j))$ is incurred by falsifying exactly one of the two clauses $(v_i \vee v_j)$ and $(\neg v_i \vee \neg v_j)$. Since the solution to the MAXSAT instance minimizes the cost, the weight of edges that are not in the cut will be minimized. This maximizes the weight of the cut.

The number of variables in this encoding is $|V|$ and the number of soft clauses is $2|E|$.

Hard Clause Encoding

In this alternative encoding of Weighted Maximum Cut, the MAXSAT instance has a variable e_{ij} for each edge $(i,j) \in E$ in addition to the variables corresponding to the vertices. If the variable e_{ij} is true, it will mean that the edge (i,j) is in cut. Hard clauses are used to ensure that the set of true variables actually corresponds to a cut in the graph, as follows. For each edge $(i,j) \in E$ there are four hard clauses that say that e_{ij} is true if and only vertices i and j are in different partitions. So $\mathit{hard}(\mathcal{F}) = \{(v_i \vee \neg v_j \vee e_{ij}), (\neg v_i \vee v_j \vee e_{ij}), (v_i \vee v_j \vee \neg e_{ij}), (\neg v_i \vee \neg v_j \vee \neg e_{ij}) : (i,j) \in E\}$. Finally, for each variable there is a soft unit clause (e_{ij}) with weight equal to the weight of the corresponding edge in the graph.

This encoding is larger than the binary clause encoding in terms of both the number of variables and clauses. There are $|V| + |E|$ variables, $|E|$ soft clauses and $4|E|$ hard clauses.

This example of encoding Weighted Maximum Cut in MAXSAT illustrates that more than one natural MAXSAT model may exist. For example, the costs may be associated only with the unit clauses, as in the hard clause encoding, or with violating non-unit clauses, as in the binary clause encoding.[2] The choice of encoding may affect the performance of state-of-the-art MAXSAT solvers.

2.3.2 Design Debugging

MAXSAT solvers can be used to find the source of errors in the design of digital circuits. In order to verify that a circuit implements the intended behaviour, various inputs to the circuit are tested. If the output of the circuit does not match the desired output, the circuit must be debugged. This means that the source of the faulty behaviour must be found and then fixed. Debugging is very time consuming for humans to perform, and can

[2]If all of the soft clauses of a MAXSAT instance are unit, then it is also a Binate Covering problem, a Pseudo-Boolean Optimization problem and an Integer Linear Program.

also be challenging to automate efficiently. MAXSAT can be used within the debugging task in order to rule out irrelevant parts of the circuit and focus on the potential error sources (Chen, Safarpour, Veneris, and Marques-Silva, 2009; Chen, Safarpour, Marques-Silva, and Veneris, 2010).

In design debugging, the circuit is translated to CNF in the usual way, by introducing a variable to represent the output of each gate, and clauses that constrain its value to be the appropriate function of its input variables. The circuit's CNF formula C is conjoined with the assignment to the input variables I that resulted in the faulty output during testing. Finally, the correct output O is also asserted, resulting in the formula $C \wedge I \wedge O$, which is unsatisfiable since the circuit represented by C does not actually produce output O when given input I. The input and output unit clauses are hard clauses, and the circuit clauses are soft with uniform weights. Therefore, the MAXSAT solution will provide a minimal subset of C that if removed will allow the circuit to be fixed. Thus the MAXSAT solution represents a possible source of error, which can be further analyzed and aggregated with other potential error sources in order to fully debug the circuit.

2.3.3 Other Applications

Another application of MAXSAT in Electronic Design Automation is FPGA (Field Programmable Gate Array) routing (Xu, Rutenbar, and Sakallah, 2003). MAXSAT solvers perform well on scheduling problems for image capture on the SPOT5 Earth observing satellite (Bensana, Lemaitre, and Verfaillie, 1999). The optimal protein alignment problem from biology was first cast as a Max Clique problem and solved efficiently using dedicated algorithms (Strickland, Barnes, and Sokol, 2005). However, the same instances when translated to MAXSAT are challenging for state-of-the-art MAXSAT solvers, as can be seen in the results of the 2009-2012 MAXSAT Evaluations on the PROTEIN-INS family (Argelich, Li, Manyà, and Planes, 2007–2012). Another application in bioinformatics is haplotype inference by pure parsimony (HIPP), which seeks to explain the genetic

makeup of a population (Graça, Marques-Silva, Lynce, and Oliveira, 2011; Graça, Lynce, Marques-Silva, and Oliveira, 2012). The winner determination problem in combinatorial auctions can be solved using Integer Programming solvers (Andersson, Tenhunen, and Ygge, 2000), but MAXSAT solvers are also effective on artificially generated instances (Leyton-Brown, Pearson, and Shoham, 2000). In planning, MAXSAT is used in optimal planning with action costs (Robinson, Gretton, Pham, and Sattar, 2010), preference-based planning (Juma, Hsu, and McIlraith, 2011), and optimizing partial-order plans (Muise, McIlraith, and Beck, 2011), as well as to learn action models from plan examples (Yang, Wu, and Jiang, 2007). The Most Probable Explanation (MPE) problem in probabilistic networks can be expressed as MAXSAT (Park, 2002). Another industrial application is upgrading software package installations (Argelich, Berre, Lynce, Marques-Silva, and Rapicault, 2010).

2.4 Resolution for MAXSAT

This section describes an algorithm for solving MAXSAT that is similar to the Davis-Putnam algorithm for SAT, that is based on ordered resolution (Davis and Putnam, 1960). The algorithm is based on the MAXRES rule, which is an equivalence-preserving transformation rule that generalizes Resolution. The MAXRES rule is defined first, followed by a description of how it can be applied by the Saturation algorithm to solve MAXSAT. Finally, the limitations of this algorithm are discussed from a practical perspective.

2.4.1 The MAXRES Rule

In propositional logic, an inference rule is a way of deriving new conclusions that are logically implied by the original formula. For example, the Resolution rule (see Appendix A) is an inference rule that can be used to derive any clause logically implied by the input

CNF formula. Inference can be thought of as transforming a formula into one that is equivalent, but in some sense more informative.

Although Resolution is sound and complete for SAT, it is *not* a sound method of inference for MAXSAT.

Example 1. *Consider the* MAXSAT *instance* $\mathcal{F} = \{(\neg x, y), (x), (\neg y)\}$ *where all clauses have weight* 1. *The clause* (y) *can be derived through Resolution on the first two clauses of* $\mathcal{F}$. *However, adding this clause changes the* MAXSAT *solutions. The truth assignment* $\pi = (x = true, y = false)$ *is a solution to* $\mathcal{F}$ *since it falsifies only one clause. On the other hand,* π *is not a solution to* $\mathcal{F} \cup \{(y)\}$*:* π *falsifies two clauses while* $\sigma = (x = true, y = true)$ *falsifies only one.*

The MAXRES rule is similar to Resolution but is sound and complete for MAXSAT (Bonet, Levy, and Manyà, 2007). MAXRES is a transformation rule, in that it removes some clauses from the formula and replaces them with another set of clauses. This is in contrast to Resolution, which only adds clauses.

MAXRES is applied to two weighted clauses (x, A) and $(\neg x, B)$ with $wt((x, A)) = w_1$ and $wt((x, B)) = w_2$, where A and B are possibly empty sets of literals. Let m be the minimum of w_1 and w_2, and define an ordering on the literals in A and B so that $A = \{a_1, .., a_k\}$ and $B = \{b_1, ..., b_j\}$. MAXRES removes the clauses above the horizontal line in Figure 2.1 from the formula and replaces them by the clauses appearing below the line. The clauses containing negated a_i and b_i variables are called the *compensation* clauses. Here, if any weight (including infinity) is subtracted from an infinite weight, the result is always infinity. Note that if A is empty, none of the compensation clauses containing $\neg a_i$ are really generated, and similarly if B is empty. Any tautologies or 0-weight clauses that would be generated are omitted. Also, if one of the clauses is subsumed by another hard clause generated in this MAXRES step, the subsumed clause will be omitted. This means that if both input clauses are hard, MAXRES behaves exactly the same as Resolution.

Clause	Weight
(x, A)	w_1
$(\neg x, B)$	w_2
(A, B)	m
(x, A)	$w_1 - m$
$(\neg x, B)$	$w_2 - m$
$(x, A, \neg b_1)$	m
$(x, A, b_1, \neg b_2)$	m
.	
.	
$(x, A, b_1, ..., b_{j-1}, \neg b_j)$	m
$(\neg x, B, \neg a_1)$	m
.	
.	
$(\neg x, B, a_1, ..., a_{k-1}, \neg a_k)$	m

Figure 2.1: The MAXRES rule. $A = \{a_1, ..., a_k\}$, $B = \{b_1, ..., b_j\}$, $k, j \geq 0$, and m is the minimum of w_1 and w_2. MAXRES replaces the two weighted clauses above the line by the set of weighted clauses appearing below the line.

It is easy to verify that the MAXRES rule is sound for MAXSAT, by showing that it preserves the cost of every truth assignment. This guarantees that applying MAXRES does not change the MAXSAT solutions.

Proposition 1. *(Bonet et al., 2007) If $\mathcal{F}$ is a* MAXSAT *instance and $\mathcal{F}'$ is the result of applying one* MAXRES *step to $\mathcal{F}$, then for all truth assignments π, $cost(\pi, \mathcal{F}) = cost(\pi, \mathcal{F}')$.*

2.4.2 The Saturation Algorithm

The MAXRES rule is potentially useful because it can simplify the MAXSAT formula and reveal more information about the optimal solution. In fact, it is possible to solve MAXSAT by using the MAXRES rule alone. That is, MAXRES is complete as well as sound for MAXSAT.

Theorem 2. *(Bonet et al., 2007)* MAXRES *is complete for* MAXSAT. *That is, given an*

instance of MAXSAT $\mathcal{F}$*, there is a sequence of* MAXRES *transformations starting with* $\mathcal{F}$ *that results in a* MAXSAT *theory* $\mathcal{F}'$ *such that the cost of the empty clauses in* $\mathcal{F}'$ *is equal to* $mincost(\mathcal{F})$*, and the rest of the clauses in* $\mathcal{F}'$ *are satisfiable. Furthermore, any truth assignment satisfying the non-empty clauses of* $\mathcal{F}'$ *is a solution to* $\mathcal{F}$*.*

The proof that MAXRES is complete for MAXSAT (Theorem 2) requires the following definition of *saturation*.

Definition 11 (Saturation)**.** *A* MAXSAT *theory* $\mathcal{F}$ *is saturated with respect to variable* x *if for every pair of clauses* C_1, C_2 *in* $\mathcal{F}$ *that clash on* x*,* C_1 *and* C_2 *also clash on another variable* $y \neq x$*.*

An algorithm to find a sequence of MAXRES transformations that satisfies the conditions of Theorem 2 is MAXSAT-Saturation, shown in Figure 1 (Bonet et al., 2007). MAXSAT-Saturation works as follows. Given an ordering over the variables, MAXSAT-Saturation performs MAXRES steps resolving on x_i until the formula A_{i-1} is saturated with respect to x_i. The algorithm sets aside the clauses that still contain x_i, and saturates the clauses not containing x_i by the next variable x_{i+1}. These saturation steps continue until all variables have been saturated. Bonet et al. prove that the result of MAXSAT-Saturation is a set of empty clauses whose weights sum to $mincost(\mathcal{F})$ and a satisfiable set of clauses (those that were set aside in each saturation step) whose satisfying truth assignments are solutions for $\mathcal{F}$. Bonet et al. show how a truth assignment that is a solution to $\mathcal{F}$ is easy to build given the sequence of set aside clauses $\{B_i\}_{i=1}^{n}$, by greedily assigning one literal at a time starting with x_n.

The worst-case time complexity of MAXSAT-Saturation is $O(m2^n)$, where m is the number of clauses in $\mathcal{F}$ and n is the number of variables. Indeed, Algorithm 1 is unlikely to work well in practise. In the worst case one MAXRES step increases the number of clauses in the formula by $\|A\| + \|B\| + 1$, or $O(n)$ where n is the number of variables. Furthermore, the clauses produced by MAXRES may be quite long. It is likely that the space required by MAXSAT-Saturation will be prohibitive in practise.

Algorithm 1: The Saturation algorithm for solving MAXSAT. $\mathcal{F}$ is a MAXSAT formula. Returns $mincost(\mathcal{F})$.

```
1 MAXSAT-Saturation (F)
2 A_0 = F
3 for i=1 to n do
4 |   S = Saturate(A_{i-1}, x_i)
5 |   A_i = {C ∈ S : x_i ∉ C}
6 |   B_i = S \ A_i
  /* Any truth assignment that satisfies ⋃_{i=1}^{n} B_i is a solution to the MAXSAT
     instance F.                                                              */
  /* A_n contains only empty clauses.                                         */
7 return Σ_{C∈A_n} wt(C)
```

Yet the MAXRES rule is used in practical MAXSAT solving. MAXRES is utilized by state-of-the-art Branch and Bound MAXSAT solvers to transform the current formula at each node of the search tree (Heras, Larrosa, and Oliveras, 2008; Larrosa, Heras, and de Givry, 2008; Li, Manyà, Mohamedou, and Planes, 2009; Kügel, 2010). The goal of applying MAXRES during Branch and Bound is to create more empty clauses, since they provide a lower bound on the cost of a solution to the current formula. The transformation may also simplify the current formula, for example, by reducing the number of clauses. However, the MAXRES rule can quickly produce many large clauses that are not necessarily useful, so in practise, MAXRES is only applied in particular cases where it is found to pay off.

2.5 Existing MAXSAT Solvers

The existing MAXSAT solvers can be divided into two groups based on their underlying algorithm: the Branch and Bound solvers, and the sequence of SAT instance solvers.

2.5.1 Branch and Bound Solvers

Branch and Bound algorithms are a common approach for solving optimization problems, including MAXSAT. Branch and Bound searches the binary tree of partial assignments to

the variables, starting with the empty assignment at the root. At each node of the tree an uninstantiated variable is chosen to branch on. The children of this node correspond to assigning the variable to *true* and *false*. This tree is explored in a depth-first manner, to find a complete assignment with minimal cost.

However, it may be unnecessary to visit all 2^n complete assignments. A first step to pruning the search space is to keep track of the cost already incurred by the partial assignment at the current node. If this known cost meets or exceeds the cost UB of the best complete assignment found so far, then the subtree below this node can be pruned. Furthermore, if it can be anticipated that some additional cost must be incurred by any complete assignment below the current node then that information can also be used to backtrack. A method of proving that any complete assignment extending a partial assignment has cost at least C is called a *lower bound* function, and it is a significant factor in the performance of Branch and Bound algorithms.

Algorithm 2 shows a recursive version of this basic algorithm. The initial call to MAXSAT-B&B-1 requires an upper bound on the cost of the MAXSAT solution, which is usually supplied by a local search procedure that tries to find a complete assignment with small cost (Tompkins and Hoos, 2004). On line 2, it is assumed that the LowerBound($\mathcal{F}$) function returns a value at least as large as the sum of the weights of the empty clauses in $\mathcal{F}$. On Line 5, SumEmpty is a function that returns the sum of the weights of the empty clauses in the given formula. On Line 6, ChooseLiteral is a function that returns a literal (whose variable appears in the given formula) according to the variable and value ordering heuristics. This chosen literal is also sometimes called the *branching* literal or the *decision* literal. Most MAXSAT solvers use a variable ordering heuristic that favours variables that appear in many clauses, many short clauses, or clauses with large weight (Wallace and Freuder, 1996; Alsinet, Manyà, and Planes, 2003; Xing and Zhang, 2005). On lines 7 and 8, the chosen literal and then its negation are instantiated, with the reduced formulas given to the recursive calls.

Algorithm 2: The basic Branch and Bound algorithm for solving MAXSAT. $\mathcal{F}$ is a MAXSAT formula. On the initial call, $UB = cost(\pi)$ for some complete assignment π. Returns $mincost(\mathcal{F})$ if it is less than UB, and UB otherwise.

```
1 MAXSAT-B&B-1 (F, UB)
2 if LowerBound(F) ≥ UB then
3     return UB
4 if F contains no variables then
5     return SumEmpty(F)
6 v = ChooseLiteral(F)
7 UB = MAXSAT-MAXSAT-B&B-1 (F|_v, UB)
8 return MAXSAT-MAXSAT-B&B-1 (F|_¬v, UB)
```

It is important for the Branch and Bound search to try to prune the part of the search space where the hard clauses of $\mathcal{F}$ can not be satisfied, since this can drastically reduce the size of the search space that must be visited. MAXSAT solvers can apply Unit Propagation (UP), and clause learning (see Appendix B) to the *hard* clauses in $\mathcal{F}$ in order to perform this type of pruning soundly and efficiently (Argelich and Manyà, 2007). MAXSAT solvers use the two watched literal data structure, first developed for SAT solvers (Moskewicz, Madigan, Zhao, Zhang, and Malik, 2001), in order to support efficient UP and backtracking (see Appendix B) (Argelich and Manyà, 2005). The VSIDS variable ordering heuristic, also borrowed from SAT solving, works well for MAXSAT solvers that learn hard clauses (Heras, Larrosa, and Oliveras, 2007).

The first Branch and Bound solver for MAXSAT was developed by Wallace and Freuder in the mid 1990s (Wallace and Freuder, 1996). Over the years, the most significant improvements have been gained by better lower bounds and the application of additional transformation rules (i.e. on lines 7 and 8 of Algorithm 2 after instantiating the chosen literal).

Lower Bounds Using Disjoint Cores

Wallace and Freuder introduced the Inconsistency Count lower bound, that only takes into account the *unit* clauses in the current formula. The sum over all variables v of

the minimum of the number of unit clauses (v) and the number of unit clauses $(\neg v)$ is a lower bound on the optimal number of falsified clauses: LB $= \#Empty(\mathcal{F}) + \Sigma_v(min\{ic(v), ic(\neg v)\})$ (Wallace and Freuder, 1996).

Note that each pair of conflicting unit clauses is actually a core of the current formula, which in this context is usually referred to as an *inconsistent subformula*. The correctness of the Inconsistency Count lower bound is based on the fact that every truth assignment will falsify either (v) or $(\neg v)$, independently of all other variables.

This idea of finding disjoint inconsistent subformulas can be extended to finding disjoint inconsistent sets each of which may contain *more* than two clauses (Li, Manyà, and Planes, 2005, 2006; Darras, Dequen, Devendeville, and Li, 2007). Unit propagation is used to find such inconsistent sets, because it can be implemented efficiently and undone very easily (see Appendix B). Given the current formula $\mathcal{F}$ at an internal node of the search, the disjoint inconsistent sets are found by applying unit propagation until a conflict is found, removing the involved clauses, and repeating UP until no more conflicts can be found in this way. The lower bound is then the sum of the weights of the minimum weight clauses from each inconsistent set. Once the lower bound has been calculated, all of the changes to the formula are undone before continuing search.

The number of disjoint inconsistent sets that will be found is dependent on the order in which unit clauses are propagated, and propagating the most recently created unit clause seems to work best in practise (Li et al., 2006). Additional inconsistent sets can be found by applying Failed Literal Detection,[3] since if unit propagation finds a conflict in both $\mathcal{F} \cup \{\ell\}$ and $\mathcal{F} \cup \{\neg\ell\}$, the set of clauses involved (excluding (ℓ) and $(\neg\ell)$), is an inconsistent subformula of $\mathcal{F}$.

One drawback to such lower bound functions is that the same inconsistent subformulas may be rediscovered at descendent nodes in the search. This is very likely to happen because when the next decision literal is instantiated, only some limited transformations

[3]Failed Literal Detection on a literal x is performed by temporarily adding the unit clause (x) to the current formula $\mathcal{F}$ and then applying UP, i.e., UP is applied to $\mathcal{F} \cup \{(x)\}$.

are applied to the formula as a result of this new variable assignment. The changes to the formula are limited because unit propagation can not be soundly applied to the soft clauses. Therefore, many of the clauses remain unchanged when a new decision is instantiated. In order to reduce the amount of recomputation, some of the inconsistent sets can be memorized (Darras et al., 2007). However in the next section we see a more popular method of making the lower bound computation incremental, based on MAXRES transformations.

Transformations and Lower Bounds

The best performing Branch and Bound solvers in the latest MAXSAT Evaluation combine a lower bound based on disjoint inconsistent subformula detection with sound transformations based on MAXRES.

The main idea is to use UP to find an inconsistent subformula R, and then apply some series of MAXRES steps to the clauses in R. The goal is to generate a new empty clause, since it will immediately contribute to the lower bound and also record the inconsistency for all descendent nodes. Furthermore, if this process is repeated until UP can find no more conflicts, the resulting lower bound may be greater than the lower bound that can be calculated using disjoint inconsistent subformulas. Intuitively, the inconsistent subformulas no longer have to be disjoint.

The challenge with this approach is that MAXRES can become expensive to apply, since in general each MAXRES step adds many compensation clauses to the formula. Therefore, several ways to apply MAXRES to inconsistent subformulas have been proposed.

For example, MINIMAXSAT (Heras et al., 2007, 2008) takes the inconsistent set found by UP and applies the analysis algorithm used to learn clauses in SAT (Algorithm 13 in Appendix B), except that each Resolution step in the derivation is replaced by a MAXRES step. It is possible to use MAXRES to copy the derivation of the learnt clause because each

clause is used no more than once. However, Heras et al. found that this transformation only pays off if the intermediate clauses generated by Algorithm 13 are all of length three or less.

All other recent Branch and Bound MAXSAT solvers are based on MAXSATZ (Li, Manyà, and Planes, 2007), including WMAXSATZ-2009 (Li et al., 2009), INCMAXSATZ (Lin, Su, and Li, 2008), WMAXSATZ+ (Li, Manyà, Mohamedou, and Planes, 2010), AKMAXSAT_LS (Kügel, 2010) and IUT_RR (Ramezani and Mousavi, 2012). These solvers apply transformations if the inconsistent set of clauses contains a subformula matching one of three patterns. The first pattern is a *Chain* containing two unit clauses and $k \geq 0$ binary clauses: $\{(\ell_1), (\neg\ell_1, \ell_2), (\neg\ell_2, \ell_3), ..., (\neg\ell_k, \ell_{k+1}), (\neg\ell_{k+1})\}$. The second pattern is Cycle Resolution restricted to three variables. The third pattern is a combination of the first two patterns. In each case, the transformation rule can be justified by a series of MAXRES steps.

Note that only binary and unit clauses are involved in the transformations applied by the MAXSATZ family of solvers, which limits the strength of their lower bounds. Also, not all inconsistent sets discovered by unit propagation will match one of the specific patterns, e.g.
$\{(\ell_1), (\neg\ell_1, \ell_2), (\neg\ell_1, \ell_3), (\neg\ell_2, \ell_4), (\neg\ell_3, \neg\ell_4)\}$ does not match the Chain or Cycle patterns. Nevertheless, the best performing Branch and Bound solver in the 2011 MAXSAT Evaluation was AKMAXSAT_LS, that uses just these rules (Kügel, 2010).

Other Lower Bounds

A few other distinct lower bounds have been proposed. One translates the current MAXSAT formula $\mathcal{F}$ to an Integer Program, and then solves its linear relaxation which gives a lower bound on $mincost(\mathcal{F})$ (Xing and Zhang, 2005).[4] Another lower bound requires that the original MAXSAT CNF $\mathcal{F}$ be relaxed by variable-splitting until its treewidth

[4]This solver also uses a unique inference rule based on a non-linear formulation of MAXSAT.

is less than 8. The relaxed CNF $\mathcal{F}'$ is then compiled to Deterministic Decomposable Negation Normal Form (d-DNNF), which allows $mincost(\mathcal{F}')$ to be calculated in time linear in the size of the d-DNNF formula. During search, the d-DNNF formula can be conditioned on the current partial assignment to get a lower bound for that node (Pipatsrisawat and Darwiche, 2007). The MHET method (Hsu and McIlraith, 2010) is related to the soft arc consistency notions studied in Weighted Constraint Satisfaction (Cooper, de Givry, Sanchez, Schiex, Zytnicki, and Werner, 2010).

Preprocessing

Transformations similar to those that are applied at each node of a Branch and Bound search tree can also be performed prior to search as a preprocessing step. For example, some early MAXSAT solvers applied an ordered form of Neighbourhood Resolution to binary clauses (Alsinet, Manyà, and Planes, 2004), or deleted pure literals as a preprocessing step (Zhang, Shen, and Manyà, 2003). Preprocessing was studied in more depth in the context of approximate MAXSAT solvers (Heras and Bañeres, 2010). However, preprocessing is not a component of any state-of-the-art exact MAXSAT solver. Therefore it is a promising direction to explore in order to tackle MAXSAT instances that remain challenging for existing approaches.

Performance of Branch and Bound MAXSAT Solvers

The Branch and Bound solvers were developed for random 2CNF and 3CNF formulas, as well as for applications like Max-Cut, Max-Clique, and Graph Colouring ("Crafted" instances). In the MAXSAT Evaluations, a Branch and Bound solver has always won in all the Random and Crafted categories. However, Branch and Bound solvers can solve very few of the Industrial category instances. The reason for this has not been adequately explained, although it must either be due to the size of the search space explored or the time taken at each node, or a combination of both factors. It should be possible to

develop more effective lower bounds for Industrial instances.

2.5.2 Sequence of SAT Instance Solvers

Another approach to solving MAXSAT is to convert the problem to a sequence of SAT instances. Each SAT instance in the sequence can be solved using a state-of-the-art SAT solver. Therefore, any improvement to SAT solvers immediately benefits MAXSAT solvers based on this approach. These MAXSAT solvers can handle some large instances from industrial applications that are too challenging for existing Branch and Bound algorithms (Argelich et al., 2007–2012).

Using the MAXSAT Decision Problem

The most obvious sequence of SAT instances can be described as follows. Given an unweighted MAXSAT instance, it is first determined if there is an assignment that falsifies zero clauses. If not, it is determined if there is an assignment that falsifies only one clause. This process is repeated, each time increasing the number of allowed falsified clauses, until the answer is 'yes', at which point the minimum number of falsified clauses has been determined. The decision problem posed at each stage is simply an instance of the decision version of MAXSAT, which can be encoded as SAT since it is in NP. Each SAT instance can be solved by using a state-of-the-art SAT solver.

This algorithm is shown in Algorithm 3. The MAXSAT decision problem is encoded as SAT by first adding a distinct relaxation variable b_i to each of the soft clauses (line 4). These new variables are called relaxation variables because setting one *true* causes its clause to be relaxed (i.e., immediately satisfied). At each iteration, a linear constraint that limits how many clauses are relaxed is translated to CNF and added to the SAT instance (line 7). The upper bound on the cost of the relaxation, k, is increased by the smallest weight in the input formula (not necessarily 1), until it equals $mincost(\mathcal{F})$, at which point the SAT instance will finally be satisfiable, terminating the loop.

There are many ways to translate the linear constraint to CNF (line 7). Some encodings introduce new variables. The size of the CNF encoding of the linear constraint can make the SAT instances harder to solve. Xu et al. proposed an encoding of the $\Sigma_i b_i \leq k$ constraint based on a linear circuit of m incrementers, one for each relaxation variable. The CNF encoding of this circuit adds $O(mlog(k))$ variables and $O(mlog^2(k))$ clauses to the original MAXSAT theory. This encoding is sufficient for instances with a small number of soft clauses, and instances with a small optimum (Xu et al., 2003).[5] Later improvements aimed for an encoding that was mostly independent of the bound k, in order to allow more learnt clauses to be saved from one iteration to the next (Fu and Malik, 2006).

Fu and Malik were also the first to suggest that refuting the augmented SAT instances might be more difficult than refuting the original formula. They note that the adder circuits, used to implement the cardinality constraints, include many XOR gates. Since unit propagation does not derive very useful information from clauses encoding XORs, the SAT solver's performance may be degraded. They also pointed out that a binary rather than linear search over k should be used when the MAXSAT solution is larger than $log(m)$. However, the instances they experimented with did not challenge the limits of the simple sequence of SAT approach.

There is also an opposite approach that starts with *satisfiable* SAT instances and works towards an unsatisfiable instance. That is, starting with an upper bound on the optimum (trivial or obtained by local search) the MAXSAT solver decreases the allowed weight of falsified clauses until an unsatisfiable instance is found. In this case, it is possible to take advantage of the solutions returned by the SAT solver to reduce the number of iterations. The cost of the satisfying truth assignment π (ignoring the relaxation variables) returned by the SAT solver can be used as the next value of k in the constraint

[5] Xu et al. applied Algorithm 3 to FPGA routing problems that had a small number of soft clauses, and unsatisfiable SAT benchmarks from the DIMACS repository that while containing only soft clauses, had optimums of at most 4.

Algorithm 3: An algorithm to solve MAXSAT using a sequence of instances of the MAXSAT Decision Problem. The input $\mathcal{F}$ is a MAXSAT formula, and the return value is $mincost(\mathcal{F})$.

```
1 MAXSAT-seq-1 (F)
2 m = min{wt(C) : C ∈ F}
3 k = −m
  /* Add a new relaxation variable to every soft clause in F */
4 F = {C_i ∪ {b_i} : C_i ∈ soft(F)} ∪ hard(F)
5 repeat
6 |   k = k + m
7 |   F_k = F ∪ CNF(Σ_i b_i wt(C_i) ≤ k)
8 until SAT-Solver(F_k) returns SAT
9 return k
```

$\Sigma b_i w_i \leq k$. This SAT→UNSAT approach is very effective on some instances, and was utilized by the winner of the Industrial Partial MAXSAT category in the 2011 MAXSAT Evaluation, QMAXSAT (Koshimura, Zhang, Fujita, and Hasegawa, 2012).

However, a drawback to these two algorithms is that the SAT instances can become very hard to solve because of the added variables and constraints. This is especially true for large instances with few hard clauses, since the number of variables in the SAT instances will be even larger than the number of soft clauses in the MAXSAT theory. Furthermore, if $mincost(\mathcal{F})$ is large in comparison to the minimum weight, or a good upper bound on the optimum can not be found, the number of iterations (i.e., SAT solving episodes) required can be prohibitive unless binary search is used.

Several alternative sequence of SAT algorithms were developed to address these issues, for example by using simpler linear constraints, reducing the number of relaxation variables, reducing the number of iterations required, or eliminating the linear constraints entirely. These algorithms are described next.

Relaxing Cores

When given an unsatisfiable CNF formula, some modern SAT solvers can output a core (see Section A for the definition of core) with not much more work than they use to

refute the theory. Although there is no guarantee that the core will be a strict subset of the original clauses, in practise it usually is. An optimal truth assignment for a MAXSAT formula will only falsify a clause if it appears in some core of the formula; this follows from Proposition 2 in Chapter 3.[6] Therefore, only clauses that belong to some core may need to be relaxed. On the other hand, at least one clause in every core *will* need to be relaxed. Therefore the number of relaxation variables used by a sequence of SAT approach can be reduced, by only adding them to clauses that appear in cores.

The first sequence of SAT algorithm to exploit this insight was Fu and Malik's Diagnosis algorithm (Fu and Malik, 2006), whose extension to Weighted Partial MAXSAT is shown in Algorithm 4 (Manquinho, Marques-Silva, and Planes, 2009; Ansótegui, Bonet, and Levy, 2009). The algorithm works by finding and "blocking" cores of the working formula. In every execution of the loop on line 5, the SAT solver is called on the current CNF formula. If the formula is satisfiable, the loop terminates and the cost of the optimum is returned on line 13. Otherwise, the SAT solver returns a core of the current formula. The formula is updated by splitting each soft clause that appears in the core into two copies, one whose weight is decreased by the minimum cost m of a clause in the core (discarding it if its weight is decreased to zero), and the other with weight m and containing a new relaxation variable (line 9). This update can be thought of as blocking m copies of the unweighted version of the core. Finally, a hard constraint saying one of these relaxation variables must be *true* is added on line 11. Note that a clause can eventually contain more than one relaxation variable, if the clause appears in more than one of the cores.

There are two advantages to Algorithm 4 compared to Algorithm 3. First, each of

[6]Proposition 2 shows that $mincost(\mathcal{F})$ is equal to the cost of a minimal cost hitting set (MCHS) of the cores of $\mathcal{F}$. Consider a truth assignment π that falsifies a clause c that does not appear in any core of $\mathcal{F}$. We show that π can not be a MAXSAT solution, as follows. It is clear that π must falsify at least one clause in every core of $\mathcal{F}$ (since each core is unsatisfiable). Thus the set of clauses falsified by π, hs, is a hitting set of the cores. Consider the set of clauses $hs' = hs \setminus \{c\}$. Since c does not appear in any core of $\mathcal{F}$, hs' is also a hitting set of the cores. But $cost(hs') < cost(hs) = cost(\pi)$. Therefore $cost(\pi)$ is greater than the cost of the MCHS of the cores. So by the proposition, π is not an optimal assignment.

the added constraints is over a smaller set of variables (just those appearing in the core). Second, the cardinality constraints, whose coefficients are all equal to one, are easier to encode as CNF than the arbitrary linear constraints used in Algorithm 3. However, the formulas at successive iterations also get larger and larger because Algorithm 4 must duplicate soft weighted clauses appearing in the cores.

Several different encodings of the cardinality constraints have been proposed, that can significantly affect the performance of Algorithm 4. Fu and Malik used the basic quadratic encoding of the constraint, which uses one clause to say that at least one variable is *true*, and a binary clause for each pair of variables saying at least one of them is *false* (Fu and Malik, 2006). Marques-Silva and Planes chose a more efficient BDD-based encoding, that adds only a linear number of variables and clauses (Marques-Silva and Planes, 2007; Eén and Sörensson, 2006). Later, a bitwise encoding was tried, that adds only $O(log(n))$ variables but $O(nlog(n))$ clauses to encode a constraint over n variables (Marques-Silva and Manquinho, 2008). The original WPM1 solver of Ansótegui et al. (Ansótegui et al., 2009) uses a linear encoding based on signed CNF (Ansótegui and Manyà, 2004).

There is an encoding of arbitrary pseudo-boolean constraints to CNF that is of polynomial size and on which unit propagation achieves Generalized Arc Consistency (GAC) (Bailleux, Boufkhad, and Roussel, 2009), however, to our knowledge it has not been applied in any MAXSAT solver. An alternative to encoding the linear constraints as CNF is to handle them natively using a Pseudo-Boolean solver (Manquinho et al., 2009).

The solver WPM1 (Ansótegui et al., 2009), based on Algorithm 4, won the Industrial unweighted and weighted partial MAXSAT categories in the 2011 MAXSAT Evaluation.

Other Algorithms that Add Constraints

Many other sequence of SAT algorithms have been proposed that are similar to Algorithms 3 and 4. They all involve adding relaxation variables to soft clauses appearing in cores of the working formula, along with linear inequality constraints on these variables.

Algorithm 4: The Diagnosis algorithm of Fu and Malik, extended to weighted partial MAXSAT. $\mathcal{F}$ is a MAXSAT formula. Returns $mincost(\mathcal{F})$.

```
1  MAXSAT-seq-2 (F)
   /* Split the formula into hard and soft clauses                    */
2  F_H = hard(F)
3  F_S = soft(F)
4  cost = 0
   /* The SAT solver ignores the clause weights                       */
5  while R = SAT-Solver(F_H ∪ F_S) returns a core do
6      B = ∅
       /* Find a minimal weight clause in the core                    */
7      m = min{wt(C) : C ∈ R}
8      for C ∈ soft(R) do
           /* Reduce the weight of C by m                             */
9          wt(C) = wt(C) − m
           /* Create a new clause that is a copy of C, relaxed by a new variable. */
10         C' = C ∪ {b}
           /* The weight of the new clause is m                       */
11         wt(C') = m
12         F_S = F_S ∪ {C'}
13         B = B ∪ {b}
       /* Add a hard constraint that one of the new relaxation variables must be
          true                                                        */
14     F_H = F_H ∪ CNF(Σ_{b∈B} b = 1)
15     cost = cost + m
16 return cost
```

msu3 is an algorithm for unweighted MAXSAT (Marques-Silva and Planes, 2007). It begins with a preliminary phase that finds as many disjoint cores of the original MAXSAT formula as possible. The number of disjoint cores provides a lower bound on the optimum. The second phase is similar to Algorithm 3 except it can begin with k equal to the number of disjoint cores found, and relaxation variables are only added as needed to the clauses appearing in the cores.

msu4 is also restricted to unweighted MAXSAT (Marques-Silva and Planes, 2008). The algorithm adds ≥ 1 constraints to relax each core it finds, until the formula becomes

satisfiable. When the formula becomes satisfiable, the number of satisfied relaxation variables in the solution is used to strictly upper bound the total number of relaxation variables that can be *true*. The algorithm alternates between series of UNSAT and SAT instances, until it terminates when the number of cores found equals the number of relaxation variables satisfied by a solution.

PM2 is an algorithm for partial MAXSAT (Ansótegui et al., 2009) that is like Algorithm 3 but also adds a new $\geq j$ constraint for each core R where j equals the number of previously found cores that are subsets of R.

WPM2 solves weighted partial MAXSAT (Ansótegui, Bonet, and Levy, 2010). Its distinguishing feature is that the discovered cores are grouped into "covers", which are a decomposition of the cores into disjoint sets. The relaxation variables in each cover are constrained to relax a particular weight of clauses k, that is updated to be the next largest value that the clauses' weights can sum up to. Even calculating the next value of k can be expensive, since it requires solving the NP-hard Subset Sum problem and additional calls to a SAT solver. The performance of WPM2 is also affected by the structure of the cores, since in the worst case all cores will eventually belong to the same cover.

WMSU1-ROR is a modification of Algorithm 4, that attempts to avoid adding relaxation variables by applying MAXRES to transform the core instead (Heras and Marques-Silva, 2011). Given the core R returned by the SAT solver, a Resolution refutation is calculated by a specialized tool. As much of this refutation as possible is copied by applying MAXRES steps to the working formula $\mathcal{F}$. The result is a transformed formula $\mathcal{F}'$, and a core of $\mathcal{F}'$, R', that is easily obtained from R. If the transformation derived the empty clause, it means the core is trivial and the sequence of SAT algorithm can continue without adding any relaxation variables for this step. Otherwise, the core R' is

relaxed as in Algorithm 4 before the next iteration begins.

WPM1-BSD (Ansótegui, Bonet, Gabàs, and Levy, 2012) and **PAR** (Martins, Manquinho, and Lynce, 2012b) both modify Algorithm 4 in a similar manner. Their idea is to only allow the SAT solver to find cores over a restricted subset of the original MAXSAT problem, increasing the size of this subset when no more cores can be found. The restricted subset can be chosen in different ways, but the most promising approach is to choose the clauses with largest weights first. This ensures that cores whose min weight clause is of largest weight are found first, which may significantly reduce the overall number of cores needed by Algorithm 4.

BINCD is a state-of-the-art solver for weighted partial MAXSAT (Heras, Morgado, and Marques-Silva, 2011; Morgado, Heras, and Marques-Silva, 2012). The BINCD algorithm is similar to WPM2, since in BINCD intersecting cores are also organized into disjoint covers. However, BINCD maintains both a lower and upper bound on the cost of each cover and performs a binary search on this cost by testing the midpoint value (Heras et al., 2011). This is in contrast to WPM2 where each cover only has a lower bound, that is successively increased to the next possible larger value. In (Morgado et al., 2012), BINCD is improved by using a global upper bound, and as a consequence the upper bounds that the original version of BINCD maintained for each cover become instead estimates of the cost that the cover contributed to the global upper bound. Another improvement is that when the lower bound for a cover is increased, a Subset Sum calculation is used to find the next possible larger value as in WPM2. Morgado et al. also proposed to use a *biased* binary search where instead of testing the middle value between the bounds, the value between the bounds is chosen based on the ratio between the number of calls to the SAT solver that have answered SAT and the number of calls that have reported UNSAT. Their policy will choose a value that is closer to the lower bound if the SAT solver has

returned a majority of SAT answers in the computation so far, and a value closer to the upper bound if the SAT solver has returned a majority of UNSAT answers.

2.6 Conclusion

This section has described the main recent approaches for solving the MAXSAT problem. Exact algorithms for MAXSAT can be divided into those that use a Branch and Bound search, and those that pose a sequence of SAT queries. In both cases, the algorithms need to reason with intersecting cores of a CNF formula. So far, two techniques have been used to deal with interesting inconsistencies: MAXRES and adding linear constraints over relaxation variables. The MAXRES rule is difficult to apply in practise because it can easily increase the size of the formula without making progress. Therefore, in practise it is only applied to particular patterns of cores involving short clauses, which limits its power and applicability. On the other hand, the technique of adding relaxation variables and constraints can create CNF formulas that are very hard for SAT solvers to refute. In the remainder of this thesis we explore a new approach, offering greater power and flexibility to reason about the cores of a MAXSAT instance.

Chapter 3

Solving MAXSAT with Hitting Sets

3.1 Introduction

Section 2.5.2 described existing MAXSAT solvers based on a sequence of SAT approach, that convert the optimization problem into a sequence of decision problems, each of which is then encoded as a SAT problem and solved with a modern SAT solver. This approach is very successful when only a few decision problems must be posed before a solution is found. However, the SAT decision problems can be much larger than the original MAXSAT instance, and performance can be significantly degraded as larger and larger decision problems must be solved.

This chapter introduces a new approach for solving MAXSAT that also utilizes a sequence of SAT problems, but in contrast to previous algorithms, the SAT problems become progressively easier. In particular, the SAT solver is only ever asked to solve problems that are composed of a *subset* of the clauses of the original MAXSAT problem.

The new approach decomposes the MAXSAT problem into two parts. One part computes minimum cost hitting sets, while in the other part the SAT solver tests the satisfiability of subsets of the original problem. This decomposition allows the SAT solver to deal only with the logical structure of the original problem. Furthermore, the sequence

of satisfiability problems that have to be solved can only become easier. However, the hitting set computations can and do become harder. It is hoped that splitting the problem in this manner will more effectively exploit the strengths of modern SAT solvers as well the strengths of solvers that are effective at performing the optimization required, e.g., integer programming solvers.

3.2 The MaxHS Algorithm

The cores of a MAXSAT instance play an important role in determining the MAXSAT solution. We have already seen that inconsistent clause subsets (i.e., cores) are used to derive lower bounds in Branch and Bound MAXSAT solvers. Recall that the Diagnosis algorithm of Fu and Malik (Algorithm 4 on page 33) solves MAXSAT by finding and relaxing cores until no additional cores can be found. In this section we formalize the connection between cores and the MAXSAT solution, showing that any solution to a MAXSAT instance must falsify at least one clause from every core. In other words, the clauses falsified by the MAXSAT solution will form a minimum cost *hitting set* of the cores. We then show how this hitting set connection can be used to guide the discovery of cores within a complete algorithm for MAXSAT.

We begin by defining the minimum cost hitting set problem (MCHS), which is a well-studied NP-hard optimization problem. We are interested in the MCHS problem because, as we will show, solving MAXSAT can be decomposed into solving a series of MCHS and SAT problems.

Definition 12 (The Minimum Cost Hitting Set Problem)**.** *Let $\mathcal{K} = \{\kappa_1, \kappa_2, ..., \kappa_k\}$ be a collection of k finite sets where each κ_i is a subset of a universe of positively weighted elements. Then a hitting set of $\mathcal{K}$ is a subset of elements hs such that $hs \cap \kappa_i \neq \emptyset$ for each $1 \leq i \leq k$. The cost of a set of elements is the sum of their weights. A minimum cost hitting set of $\mathcal{K}$ is a hitting set of $\mathcal{K}$ such that there is no other hitting set of smaller*

cost.

We will now explain how the MCHS problem can be used to solve MAXSAT. As mentioned in Section 2.2, it is assumed that the hard clauses of the MAXSAT instance are satisfiable, and that the cost of the MAXSAT solution is greater than zero. Under these conditions, it is easy to observe that every truth assignment necessarily falsifies at least one clause in every core of the MAXSAT instance, and therefore, every truth assignment corresponds to a hitting set of the instance's cores. Thus the cost of the MAXSAT solution will be at least the cost of the *minimum cost* hitting set of the cores. This raises the question of whether there exists a truth assignment that actually achieves this minimum cost. The answer is affirmative: the MAXSAT solution always achieves this minimum cost.

Proposition 2. *If $\mathcal{F}$ is a* MAXSAT *instance, $\mathcal{K}$ is the set of all cores of $\mathcal{F}$, and hs is a minimum cost hitting set of $\mathcal{K}$, then $mincost(\mathcal{F}) = cost(hs)$.*

Proof. Let *hs* be a minimum cost hitting set of $\mathcal{K}$. Any truth assignment falsifies a set of clauses that form a hitting set of $\mathcal{K}$. Therefore, since *hs* is a MCHS of $\mathcal{K}$, $mincost(\mathcal{F}) \geq cost(hs)$. It is not possible to prove the empty clause from $\mathcal{F} \setminus hs$, otherwise there would be a core κ in $\mathcal{K}$ such that $\kappa \subseteq (\mathcal{F} - hs)$. But then *hs* would not hit κ and *hs* would not be a hitting set of $\mathcal{K}$. Thus $\mathcal{F} \setminus hs$ is satisfiable and therefore $mincost(\mathcal{F}) \leq cost(hs)$. □

Proposition 2 says that the technique of finding hitting sets can solve the MAXSAT problem: it is a complete method. However, as stated it is also quite impractical. For one there are an exponential number of possible cores of $\mathcal{F}$. Consider the case where instead of having access to all cores of $\mathcal{F}$, an incomplete collection of cores is available. In this case a minimum hitting set provides a *lower bound* on $mincost(\mathcal{F})$.

Proposition 3. *If $\mathcal{K}$ is any set of cores of* MAXSAT *instance $\mathcal{F}$, and hs is a minimum cost hitting set of $\mathcal{K}$, then $mincost(\mathcal{F}) \geq cost(hs)$. That is, $cost(hs)$ is a lower bound on the cost of the solution of $\mathcal{F}$.*

Proof. If $\mathcal{K}$ and $\mathcal{K}'$ are two sets of cores with $\mathcal{K} \subset \mathcal{K}'$, with minimum cost hitting sets hs and hs' respectively, then $cost(hs) \leq cost(hs')$, because every hitting set of $\mathcal{K}'$ must be a hitting set of $\mathcal{K}$. □

So any set of cores of a MAXSAT instance can provide a lower bound on the MAXSAT solution's cost. Can an incomplete set of cores ever be sufficient to prove the MAXSAT solution itself? If in fact an incomplete set of cores is enough, how can we know that the lower bound they provide is equal to the optimum? The next theorem gives a condition on a set of cores and their MCHS under which we can derive the MAXSAT solution.

Theorem 3. *If $\mathcal{K}$ is a set of cores for the* MAXSAT *problem $\mathcal{F}$, hs is a minimum cost hitting set of $\mathcal{K}$, and π is a truth assignment satisfying $\mathcal{F} \setminus hs$ then $mincost(\mathcal{F}) = cost(\pi) = cost(hs)$.*

Proof. $mincost(\mathcal{F}) \leq cost(\pi)$ as $mincost(\mathcal{F})$ is the minimum over all possible truth assignments. $cost(\pi) \leq cost(hs)$ as the clauses π falsifies are a subset of hs (π satisfies all clauses in $\mathcal{F} - hs$). On the other hand $mincost(\mathcal{F}) \geq cost(hs)$. Any truth assignment must falsify at least one clause from every core $\kappa \in \mathcal{K}$. Thus for any truth assignment τ, $cost(\tau)$ must include at least the cost of a hitting set of $\mathcal{K}$. This cannot be any less than $cost(hs)$ which has minimum cost. □

So any set of cores $\mathcal{K}$ is sufficient to determine the MAXSAT solution if the MAXSAT formula can be rendered satisfiable by removing a set of clauses that forms a minimum cost hitting set of $\mathcal{K}$. Thus, a SAT check provides the stopping condition for a MAXSAT algorithm that accumulates cores and calculates lower bounds based on the cores' MCHS. This idea for a new MAXSAT algorithm is formalized in Algorithm 5. The algorithm starts off with an empty set of cores $\mathcal{K}$. At each stage it computes a minimum cost hitting set hs via the function "FindMinCostHittingSet" and calls a SAT solver to determine if $\mathcal{F} \setminus hs$ is satisfiable. If it is satisfiable then the SAT solver returns $(true, \kappa)$ with κ set to a satisfying assignment for $\mathcal{F} \setminus hs$. Otherwise, if $\mathcal{F} \setminus hs$ is still unsatisfiable, the SAT solver

Algorithm 5: The MaxHS algorithm for solving MAXSAT

```
1 MAXHS-basic (F)
2 K = ∅
3 while true do
4     hs = FindMinCostHittingSet(K)
5     (sat?,κ) = SatSolver(F \ hs)
      ; // If SAT, κ contains the satisfying truth assignment.
      ; // If UNSAT, κ contains an UNSAT core.
6     if sat? then
7         break ; // Exit While Loop
      // Add new core to set of cores
8     K = K ∪ {κ}
9 return (κ, cost(κ))
```

returns (*false*, κ) with κ set to a new core of $\mathcal{F} \setminus hs$. New cores are added to $\mathcal{K}$, while satisfying assignments cause the algorithm to terminate.

Proposition 4. *Algorithm 5 correctly returns a solution to the inputted* MAXSAT *problem* $\mathcal{F}$*. That is, it returns a truth assignment* κ *for* $\mathcal{F}$ *that achieves mincost*$(\mathcal{F})$*.*

Proof. First observe that Algorithm 5 only returns when it breaks out of the while loop, and this occurs only when the current $\mathcal{F} \setminus hs$ is satisfiable. Since in this case hs is a minimum cost hitting set of a set of cores and κ is a truth assignment satisfying $\mathcal{F} \setminus hs$, by Theorem 3 $cost(\kappa) = mincost(\mathcal{F})$. This shows that the algorithm is sound.

Second, to show that the algorithm is complete, it simply needs to be shown that it must terminate. Since $\mathcal{F}$ is a finite set of clauses, the set of cores of $\mathcal{F}$ must also be finite. Each iteration of the while loop computes a new core of $\mathcal{F}$ and adds it to $\mathcal{K}$. This core cannot be the same as any previous core, hence the while loop must eventually terminate. Consider the hitting set hs computed at line 4 prior to the computation of κ at line 5. $\kappa \cap hs = \emptyset$ since $\kappa \subseteq (\mathcal{F} \setminus hs)$. However, for any previously computed core κ^- we have that $\kappa^- \cap hs \neq \emptyset$ since hs is a hitting set of all previous cores. Hence for all previous cores κ^-, $\kappa \neq \kappa^-$. □

3.3 Benefits of maxhs

The worst-case behaviour of Algorithm 5 is analyzed in Section 3.4 below, but first we point out why it is a promising basis for a new state-of-the-art maxsat solver. First, maxhs is able to immediately exploit advances in practical SAT solving, since it uses a SAT solver as a black-box. This is a benefit shared with existing maxsat solvers based on a sequence of SAT approach. However, all existing algorithms must add extra variables and constraints to the SAT instances, which may result in very challenging instances for the SAT solver. In contrast, maxhs only asks the SAT solver to deal with *subsets* of the original maxsat formula. We expect that in practise, the various subsets of the maxsat formula are likely to be of similar difficulty for the SAT solver to refute. Therefore, maxhs better exploits SAT solver technology than the existing sequence of SAT algorithms.

The second benefit of maxhs is that it allows the SAT solver to be combined with another powerful technology in a hybrid approach. maxhs decomposes the maxsat problem into a SAT problem and a MCHS problem; the latter is equivalent to the Set Cover problem.[1] The Set Cover problem, like maxsat, is an NP-hard optimization problem, but it has been well studied from a theoretical and practical perspective by the computer science and operations research communities respectively. Very large instances of Set Cover, arising in industrial applications, are routinely solved by sophisticated Mixed Integer Programming solvers like cplex. The maxhs algorithm can utilize such a MIP solver as a black-box, to solve the MCHS instances efficiently.

As we have discussed, the maxhs algorithm allows the differing strengths of SAT

[1] An instance of Set Cover is a finite set $\mathcal{K} = \{\kappa_1, ..., \kappa_m\}$ where each $\kappa_i \in \mathcal{K}$ is a finite set of elements from universe $U = \bigcup_{\kappa_i \in \mathcal{K}} \kappa_i$. There is also a weight $wt(\kappa_i) \in \mathbb{R}^{>0}$ associated with each $\kappa_i \in \mathcal{K}$. The answer to the Set Cover instance is a subset $\mathcal{K}' \subseteq \mathcal{K}$, such that $\bigcup_{\kappa \in \mathcal{K}'} \kappa = U$, that is, a set of sets whose union contains all possible elements. Furthermore, $\mathcal{K}'$ must be a minimum cost subset satisfying this property, where the cost of $\mathcal{K}'$ is given by $cost(\mathcal{K}') = \Sigma_{\kappa \in \mathcal{K}'} wt(\kappa)$. The equivalent MCHS instance is as follows. For each $e \in U$ in the Set Cover instance, there is a set $S_e = \{\kappa_i : e \in \kappa_i\}$ in the MCHS instance. Then the MCHS solution is a minimum cost set of κ_i that hits every S_e, i.e. whose union includes every $e \in U$.

solvers and MIP solvers to be combined, in a clean and effective manner. Both solvers are used as a black-box, and the problems given to each are well suited to the solvers' abilities.

3.4 Factors Affecting Performance

The worst case complexity of solving MAXSAT with a Branch and Bound solver is $2^{O(n)}$ where n is the number of variables. However, the worst case complexity of Algorithm 5 is worse. There are 2^m possible cores where m is the number of soft clauses. This provides a worst case bound on the number of iterations executed in the algorithm. Each iteration requires solving a SAT problem of $2^{O(n)}$ and a hitting set problem of $2^{O(m)}$ (one has to examine sets of clauses to find a hitting set). This leaves the worst case complexity $2^{O(m)} \times (2^{O(n)} + 2^{O(m)}) = 2^{O(m)}$ since $m > n$ (typically, the number of clauses m is much larger than the number of variables n).

Fortunately, we can say more about the expected behaviour of MAXHS than just its worst case complexity. The performance of the MAXHS algorithm is affected by three factors: the difficulty of the SAT instances, the difficulty of the MCHS instances, and the total number of cores required. These factors are not independent of each other, since as the number of cores grows, the MCHS instances get larger and therefore harder to solve. The time spent by the SAT solver may also be traded off against the other two factors. For example, more time can be spent on SAT solving in order to find *particular* cores that reduce the total number of cores required, or change the structure of the MCHS instances making them easier to solve. Thus the interplay between these three factors offers a rich space in which to explore trade-offs.

On the other hand, the performance of MAXHS is likely to be quite variable and hard to predict because of these three factors. MAXHS relies on two potentially expensive subroutines, the SAT solver and the MCHS solver, whose runtimes can not be predicted

easily. In fact, any one of the three factors can be the source of exponential runtime in MAXHS, as shown by the following three examples.

Example 2. *Let $\mathcal{F}$ be an instance of the Pigeon Hole Principle, where all clauses are considered soft with uniform weights. Note that removing any single clause from $\mathcal{F}$ will make the remaining clauses satisfiable. Therefore,* MAXHS *will terminate after the first core is found. So only one MCHS problem will be solved, and it is trivial. However, the time spent by the SAT solver to find the single core will be exponential.*

Example 3. *Given any known MCHS solver, there exists some MCHS instance for which its runtime will be exponential (unless $P = NP$). Let $\mathcal{K}$ be a MCHS instance. We construct a* MAXSAT *instance $\mathcal{F}$ that is equivalent to $\mathcal{K}$ as follows. For each set $\kappa \in \mathcal{K}$, where $\kappa = \{e_1, ..., e_k\}$, there is a hard clause $(e_1 \vee, ..., \vee e_k)$. Finally, there is a soft clause $(\neg e)$ with weight $wt(e)$ for each element $e \in \bigcup_{\kappa \in \mathcal{K}} \kappa$. A minimal core is a core such that any proper subset is not a core. It is easy to see that the minimal cores of $\mathcal{F}$ correspond to the hard clauses of $\mathcal{F}$ and therefore the total number of minimal cores is equal to $|\mathcal{K}|$. The SAT solver can find each of the minimal cores in polynomial time, by using unit propagation alone. The number of minimal cores required by* MAXHS *is at most $|\mathcal{K}|$. So the only possible source of exponential runtime on $\mathcal{K}$ is solving the MCHS problems, and assuming that $P \neq NP$, there is some MCHS instance $\mathcal{K}$ on which* MAXHS *will take exponential time.*

Proposition 5. *Let $E = \{e_1, ..., e_n\}$ be a universe of n equally weighted elements. Let $\mathcal{K}_{n,r} = \{\kappa \subset E : |\kappa| = r\}$ be an instance of the MCHS problem where $1 \leq r \leq n$. Then the MCHS of $\mathcal{K}_{n,r}$ is of size $n - r + 1$.*

Proof. By induction on n. When $n = 1$, $r = 1$ and there is only one set in $\mathcal{K}_{1,1}$. The MCHS is of size $1 = n - r + 1$. Let $n \geq 1$ and assume that the size of the MCHS of $\mathcal{K}_{n,r}$ is $n - r + 1$ for all $1 \leq r \leq n$. Now consider $\mathcal{K}_{n+1,r}$ for some $1 \leq r \leq n + 1$. If $r = n + 1$ then $\mathcal{K}_{n+1,r}$ has only one set to hit and its MCHS is of size $1 = (n + 1) - (n + 1) + 1$.

So assume that $r < n+1$. Let hs be a MCHS for $\mathcal{K}_{n+1,r}$ and without loss of generality assume that hs contains element e_{n+1}. Consider $\mathcal{K}' = \mathcal{K}_{n+1,r} \setminus \{\kappa \in \mathcal{K}_{n+1,r} : e_{n+1} \in \kappa\}$. Each element of $E = \{e_1, ..., e_{n+1}\}$ appears in exactly $\binom{n}{r-1}$ of the sets of $\mathcal{K}_{n+1,r}$, because given any element there are $\binom{n}{r-1}$ ways to choose the other $r-1$ elements to make up a set. So $|\mathcal{K}'| = |\mathcal{K}_{n+1,r}| - \binom{n}{r-1} = \binom{n+1}{r} - \binom{n}{r-1} = \binom{n}{r}$ as shown below.

$$\begin{aligned}
\binom{n+1}{r} - \binom{n}{r-1} &= \frac{(n+1)!}{(n+1-r)!r!} - \frac{n!}{(n-r+1)!(r-1)!} \\
&= \frac{(n+1)! - n!r}{(n+1-r)!r!} \\
&= \frac{n![(n+1)-r]}{(n+1-r)!r!} \\
&= \frac{n!}{(n-r)!r!} \\
&= \binom{n}{r}
\end{aligned}$$

It is easy to see that $\mathcal{K}'$ is actually equal to $\mathcal{K}_{n,r}$. Furthermore, $hs' = hs \setminus \{e_{n+1}\}$ is a hitting set for $\mathcal{K}'$, of size $|hs| - 1$. By the Induction Hypothesis and our assumption that $r < n+1$, the MCHS of $\mathcal{K}' = \mathcal{K}_{n,r}$ is of size $n - r + 1$. Thus $n - r + 1 \leq |hs'| = |hs| - 1$. So $|hs| \geq n - r + 2 = (n+1) - r + 1$. It remains to show that a hitting set of size $(n+1) - r + 1$ exists for $\mathcal{K}_{n+1,r}$. Such a hitting set can be constructed by taking a MCHS of $\mathcal{K}_{n,r}$ and adding element e_{n+1}.

□

Proposition 6. *Let n be an even number and let $E = \{e_1, ..., e_n\}$ be a universe of equally weighted elements. Let $\mathcal{K}_{n,r} = \{\kappa \subset E : |\kappa| = r\}$ be an instance of the MCHS problem where $r = \frac{n}{2}$. Let $\mathcal{K}' = \mathcal{K}_{n,r} \setminus \kappa'$ for some $\kappa' \in \mathcal{K}_{n,r}$. Then the MCHS of $\mathcal{K}'$ is strictly smaller than the MCHS of $\mathcal{K}_{n,r}$.*

Proof. By Proposition 5, the MCHS of $\mathcal{K}_{n,r}$ is of size $n - r + 1 = n - \frac{n}{2} + 1 = \frac{n}{2} + 1$. We will show that the MCHS of $\mathcal{K}'$ is of size at most $\frac{n}{2}$ and is therefore strictly smaller.

We do this by arguing that the $n - r = \frac{n}{2}$ elements that do not appear in κ' hit all sets in $\mathcal{K}'$. Assume for contradiction that there is a set $\kappa \in \mathcal{K}'$ that is not hit by $E \setminus \kappa'$. That is, assume that $\kappa \cap (E \setminus \kappa') = \emptyset$. Since $\kappa \subset E$, this means that $\kappa \subseteq \kappa'$. But this is impossible since $|\kappa| = |\kappa'|$, $\kappa \in \mathcal{K}'$ and $\kappa' \notin \mathcal{K}'$ and $\mathcal{K}'$ contains no duplicate sets. Therefore, $n - r = \frac{n}{2}$ elements are enough to hit all sets in $\mathcal{K}'$. □

Example 4. *Let $\mathcal{F}$ be a* MAXSAT *instance with an even number n of soft clauses with uniform weights, $(x_1), ..., (x_n)$ and let the hard clauses of $\mathcal{F}$ form a clausal encoding of the cardinality constraint $\Sigma_{i=1}^{n} x_i < n/2$. On this family of problems, an exponential number of cores will always be required by* MAXHS*, as we explain next. The solutions to $\mathcal{F}$ are the truth assignments that set as many of the variables to true as possible without violating the hard cardinality constraint. Thus a solution to $\mathcal{F}$ will set exactly $\frac{n}{2} - 1$ of the x_i variables to true and the rest to false, and $\frac{n}{2} + 1$ is the optimal cost. Any subset of the n soft clauses, with size greater than or equal to $\frac{n}{2}$, is a core of $\mathcal{F}$. Therefore, $\mathcal{F}$ has at least $\binom{n}{n/2}$ cores. By Proposition 6, for any number of cores $k < \binom{n}{n/2}$, the cost of their MCHS is less than the optimum. Therefore,* MAXHS *will require at least $\binom{n}{n/2}$ cores, which is exponential in n. Each core has a polynomial size refutation, assuming a suitable encoding of the cardinality constraint is used (Bailleux et al., 2009).*

We will see that in practise, the cores are often easy for a SAT solver to refute and the SAT solving time makes up a small fraction of the total time taken by MAXHS. The limiting factor is the time taken to solve the MCHS instances, which grows quickly as more cores are added. This limits the total number of iterations MAXHS can accomplish. However, the number of cores required by MAXHS does vary depending on which cores are discovered by the SAT solver. Therefore it is important to solve the MAXSAT problem in as few iterations as possible, by finding the right set of cores. Section 3.6 introduces techniques which improve the quality of the cores by encouraging core *diversity*.

In the remainder of the thesis, we take an experimental approach to investigate the trade-offs that affect the performance of MAXHS. The first step is to implement the basic

maxhs algorithm, as described in the next section.

3.5 Implementation

Algorithm 5 was implemented in C++. The two main decisions that must be made when implementing maxhs are which SAT solver to use, and how to solve the MCHS problems. We chose to use minisat-2.0 as the SAT solver, because of its good performance and extensible source code (Eén and Sörensson, 2003). minisat is based on the DPLL algorithm as described in Appendix B. For the MCHS problems, we first tried creating two special-purpose solvers, based on A^* search and Branch and Bound respectively. However, their preliminary performance was not very promising in comparison to the more general and mature MIP solver cplex. We believe that considerable ingenuity and engineering would be required to outperform cplex on the MCHS instances arising from maxhs. Therefore, in order to focus on questions raised by maxhs as a whole, we decided to use cplex to solve the MCHS problems.

3.5.1 Extracting Cores

The minisat-2.0 SAT solver is used to compute cores. Our implementation of maxhs actually incorporates the source code of the "core" solver of minisat-2.0. This offers more flexibility and efficiency than invoking the SAT solver as a separate process, as well as allowing the state of the SAT solver (e.g. the learnt clauses and activities) to persist from one SAT solving episode to the next. We use a simple trick of introducing relaxation variables to the soft clauses, which makes extracting cores very simple (Ryvchin and Strichman, 2011). This trick also means that any clauses learned by the SAT solver in one iteration are sound for all other iterations as well. Similar to other core-based maxsat solvers (see Section 2.5.2), a unique "relaxation variable" is added to each clause of $soft(\mathcal{F})$. So soft clause C_i becomes $C_i \cup \{b_i\}$ where b_i appears nowhere else in the new

theory. The hard clauses of $\mathcal{F}$ are left unchanged.

Definition 13 (b-Variable Relaxation)**.** *If $\mathcal{F}$ is a* MAXSAT *instance, then the b-variable relaxation of $\mathcal{F}$ is a CNF formula $\mathcal{F}^b = \{(C_i \vee b_i) : C_i \in soft(\mathcal{F})\} \cup hard(\mathcal{F})$.*

The CNF formula $\mathcal{F}^b$ is called the b-variable relaxation of $\mathcal{F}$ because the b-variables can be used to relax the constraints of $\mathcal{F}$. If b_i is set to *true*, the clause C_i is removed from the theory. On the other hand, if b_i is set to *false*, the clause C_i is activated.

Each solution of $\mathcal{F}^b$ has an associated cost, which is equal to the sum of the weights of the clauses corresponding to the b-variables it sets to *true*.

Definition 14 (*bcost*)**.** *If π is a truth assignment to the variables of $\mathcal{F}^b$, we define its cost as $bcost(\pi)$: if $\pi \not\models \mathcal{F}^b$ then $bcost(\pi) = \infty$, otherwise $bcost(\pi) = \sum_{b_i:\pi\models b_i} wt(C_i)$.*

The problem of finding a minimum cost solution of $\mathcal{F}^b$ is equivalent to solving the original MAXSAT instance. As shown by the following proposition, the minimum *bcost* satisfying assignments for $\mathcal{F}^b$ correspond to solutions of $\mathcal{F}$.

Proposition 7. *$mincost(\mathcal{F}) = \min_\pi bcost(\pi)$, where the minimum is taken over all truth assignments π to the variables of $\mathcal{F}^b$. Furthermore, if π achieves a minimum value of $bcost(\pi)$, then π restricted to the variables of $\mathcal{F}$ is a solution for $\mathcal{F}$.*

Proof. Let π be a truth assignment of minimum *bcost*, and let π' be π restricted to the variables of $\mathcal{F}$. Then π satisfies every clause of $\mathcal{F}^b$, since a truth assignment satisfying $hard(\mathcal{F})$ can be extended to a satisfying assignment for $\mathcal{F}^b$ by assigning all of the b-variables to *true*, and furthermore, $\sum_{b_i} wt(C_i) < \infty$ so the minimal *bcost* assignment will be one that satisfies $\mathcal{F}^b$.

Next, we show that π sets a b-variable to *true* if and only if it is necessary to satisfy the clause. Suppose for contradiction that π' satisfies a clause C_i of $\mathcal{F}$ such that π assigns the corresponding b-variable for that clause, b_i, the value *true*. Then the assignment to the variables of $\mathcal{F}^b$ that is the same as π but with b_i set to *false* has *bcost* strictly less

than that of π (since C_i remains satisfied even though the value of b_i is changed, and b_i does not appear in any other clause of $\mathcal{F}^b$.). This is a contradiction since π was chosen to have minimal *bcost*. Therefore, for every clause C_i of $\mathcal{F}$ that is satisfied by π', π assigns its b_i to *false*.

On the other hand, if π' falsifies C_i, it must be the case that b_i is set to *true* in π since π satisfies all clauses in $\mathcal{F}^b$. Thus we have shown that the b-variables that π sets to *true* correspond to the clauses that π' falsifies, and $bcost(\pi) = cost(\pi')$.

Finally, we show that $bcost(\pi) = cost(\pi') = mincost(\mathcal{F})$ and thus that π' is a MAXSAT solution. Assume for contradiction that $cost(\pi') > mincost(\mathcal{F})$. Then $bcost(\pi) = cost(\pi') > mincost(\mathcal{F})$. Let σ be a MAXSAT solution to $\mathcal{F}$. We can define a truth assignment σ' to the variables of $\mathcal{F}^b$ by extending σ to the b-variables as follows. If σ falsifies clause $C_i \in \mathcal{F}$, then σ' assigns b_i to *true*, and if σ satisfies C_i then σ' assigns b_i to *false*. Then $cost(\sigma) = bcost(\sigma')$, but $cost(\sigma) = mincost(\mathcal{F}) < bcost(\pi)$ and therefore $bcost(\sigma') < bcost(\pi)$. This contradicts the fact that π has minimal *bcost*. □

The b-variable relaxation allows any subset of $\mathcal{F}$ to be selected through the proper instantiation of the b-variables. Therefore, when Algorithm 5 needs to test the satisfiability of $\mathcal{F} \setminus hs$, the b-variables associated with the clauses in hs are set to *true*, and all other b-variables are set to *false*.

Observation 1. *Let hs be a subset of $soft(\mathcal{F})$ and let $\pi = \{b_i : C_i \in hs\} \cup \{\neg b_i : C_i \in soft(\mathcal{F}) \setminus hs\}$ be a truth assignment to the b-variables of $\mathcal{F}^b$. Then $\mathcal{F} \setminus hs$ is the same as $\mathcal{F}^b$ reduced by π, i.e., $\mathcal{F} \setminus hs = \mathcal{F}^b|_\pi$.*[2]

Proof. First we note that truth assignment π assigns a value to every b-variable in $\mathcal{F}^b$, so after $\mathcal{F}^b$ is reduced by π, only original variables will remain in $\mathcal{F}^b|_\pi$. Consider a clause in $\mathcal{F} \setminus hs$. If the clause is hard, it appears unchanged in $\mathcal{F}^b$ and after reduction by π it will still remain unchanged in $\mathcal{F}^b|_\pi$ (since it does not contain a b-variable). So consider a

[2]See Appendix A

clause C_i in $soft(\mathcal{F}) \setminus hs$. By definition of π, π assigns b_i to *false*. Therefore, the relaxed clause $C_i \cup \{b_i\} \in \mathcal{F}^b$ will be reduced to C_i by π. Thus we have shown that $C_i \in \mathcal{F}^b|_\pi$ and therefore $\mathcal{F} \setminus hs \subseteq \mathcal{F}^b|_\pi$.

For the other direction, consider a clause in $\mathcal{F}^b|_\pi$. If the clause did not contain a b-variable in $\mathcal{F}^b$ then it was a hard clause of $\mathcal{F}$, and therefore it will also appear in $\mathcal{F} \setminus hs$ since $hs \subseteq soft(\mathcal{F})$. So let $C_i \in \mathcal{F}^b|_\pi$ and assume that $C_i \cup \{b_i\} \in \mathcal{F}^b$. If π had assigned b_i to *true*, then C_i would have been satisfied by reducing $\mathcal{F}^b$ by π. So it must be the case that π assigned b_i to *false*. By the definition of π, this only occurs if $C_i \in \mathcal{F} \setminus hs$. Thus we have also shown that $\mathcal{F}^b|_\pi \subseteq \mathcal{F} \setminus hs$.

□

These b-variable assignments are added as "assumptions" in MINISAT. MINISAT provides the assumption interface to test whether a given set of literals can be extended to a satisfying assignment. MINISAT can take as input a set of assumptions $\mathcal{A}$, specified as a set of literals, along with a CNF formula $\mathcal{F}$ and then determine if $\mathcal{F} \wedge \mathcal{A}$ is satisfiable. It will return a satisfying truth assignment for $\mathcal{F} \wedge \mathcal{A}$ if one exists (this truth assignment necessarily extends $\mathcal{A}$). Otherwise it will report UNSAT and return a learnt clause C which is a disjunction of negated literals of $\mathcal{A}$. This clause has the property that $\neg C$ specifies a subset of $\mathcal{A}$ such that $\mathcal{F} \wedge \neg C$ is unsatisfiable. This means $\mathcal{F} \models C$. We give MINISAT the CNF $\mathcal{F}^b$ and the assumptions $\mathcal{A} = \pi$ as specified in Observation 1. If $\mathcal{F} \setminus hs$ is UNSAT, MINISAT will compute a conflict clause over the assumptions—the set of assumptions that lead to failure. The *true* b-variables do not impose any constraints so they cannot appear in the conflict clause. Instead, the conflict clause contains the set of *false* b-variables that caused UNSAT. The core is simply the set of clauses associated with the b-variables of the computed conflict.

Proposition 8. *If* $\mathcal{F}^b \models (b_{i1} \vee, ..., \vee b_{ik})$ *for some set of b-variables* $\{b_{ij}\}|_{j=1}^k$, *then* $\kappa = \{C_{i1}, ..., C_{ik}\}$ *is a core of* $\mathcal{F}$.

Proof. Assume for contradiction that κ is not a core of $\mathcal{F}$. Therefore, by definition $\kappa \cup \mathit{hard}(\mathcal{F})$ is satisfiable. Let π be a truth assignment to the variables of $\mathcal{F}$ that satisfies $\kappa \cup \mathit{hard}(\mathcal{F})$. We define a truth assignment π' to the variables of $\mathcal{F}^b$ that extends π as follows. If b_i corresponds to a clause $C_i \in \kappa$, then π' assigns b_i to *false*. Otherwise, if b_i does not correspond to a clause in κ, π' assigns b_i to *true*. Then π' will satisfy $\mathcal{F}^b$ and falsify $(b_{i1} \vee, ..., \vee b_{ik})$. This contradicts the fact that $\mathcal{F}^b \models (b_{i1} \vee, ..., \vee b_{ik})$, so by contradiction κ is a core of $\mathcal{F}$. □

3.5.2 Computing a Minimum Cost Hitting Set

The hitting set problem is encoded as an integer linear program (ILP) and the MIP solver CPLEX (version 12.2) is invoked to solve it. The minimum cost hitting set problem is the same as the minimum cost set cover problem and standard ILP encodings exist, e.g., (Vazirani, 2001). The encoding used in this thesis introduces a 0/1 variable x_i for each clause C_i appearing in a core; for each core there is the constraint that the sum over the x_i variables of the clauses it contains is greater or equal to 1; and the objective is to minimize the sum of $wt(C_i) \times x_i$.

$$\begin{aligned} \text{minimize: } & \sum_{i=1}^{m} wt(C_i) \cdot x_i & \\ \text{where: } & \sum_{C_i \in \kappa} x_i \geq 1 & \text{for } \kappa \in \mathcal{K} \\ & x_i \in \{0,1\} & \text{for } 1 \leq i \leq m \end{aligned}$$

3.6 Core Diversification

The number of iterations performed by MAXHS depends on the particular set of cores found. In this section we describe the two main techniques we employ to find a better

set of cores. Both techniques are inspired by the observation that the smallest possible number of iterations is achieved when all of the discovered cores are disjoint, i.e., no clause appears in more than one core. Of course, it is usually impossible to solve the entire maxsat problem with disjoint cores alone. Yet it is still desirable to find cores that intersect as little as possible.

3.6.1 Minimal Cores

The cores returned by the SAT solver are not necessarily minimal, in the sense that the cores may include extra clauses that do not contribute to the contradiction.

Definition 15 (Minimal Core)**.** *A core κ of a* maxsat *instance $\mathcal{F}$ is minimal if every proper subset of κ is not a core of $\mathcal{F}$.*

If a non-minimal core is added to the MCHS instance, the MCHS solver may choose to put one of the superfluous clauses in the hitting set. Relaxing such clauses can not fix the contradiction, so another core will be required to rule out this hitting set, increasing the number of iterations required. Therefore, in order to reduce the number of iterations we can minimize the cores returned by the SAT solver. We use a simple destructive MUS (Minimal Unsatisfiable Subset) algorithm, as described in (Silva and Lynce, 2011), to generate minimal cores. The MUS algorithm works as follows. Given a core κ, it iterates through the clauses in κ. For each clause $C \in \kappa$, it asks the SAT solver if $\{\kappa \setminus \{C\}\} \cup \mathit{hard}(\mathcal{F})$ is satisfiable, which determines if the core is still a core if C is removed. If the SAT solver reports UNSAT, clause C is removed from κ before moving on to check the next clause in the core. Since this minimization routine runs a SAT check for each clause in the core, we may expect that minimizing cores will be time consuming.[3] However, in Section 3.7 we verify that minimizing the cores does pay off in practise.

[3]The problem of finding a minimal core is D^P-complete (Papadimitriou and Wolfe, 1988). The class D^P contains the languages that are equal to the intersection of a language in NP and one in co-NP.

3.6.2 Re-refuting Cores

We consider an alternative to minimal cores, that may still reduce the size of the cores but is faster to compute. The re-refutation method tries to reduce the size of a core returned by the SAT solver by asking the SAT solver to refute the core again. In some cases the SAT solver will be able to find a different refutation of the core that uses a smaller subset of its clauses. We recurse on this smaller core, asking the SAT solver to refute it again. We continue in this way until the size of the core no longer gets smaller. The amount that the core will be minimized using re-refutation depends on the behaviour of the SAT solver, and there is no guarantee that the resulting core will be minimal. This method of reducing the size of cores was used previously (Davies and Bacchus, 2011), but we discover in Section 3.7.2 that finding minimal cores gives much better performance.

3.6.3 Disjoint Cores

When a core is discovered that is disjoint from all previously found cores, the cost of the current MCHS is guaranteed to increase. Furthermore, solving the MCHS problem is trivial for disjoint cores. These observations motivate finding as many disjoint cores as possible in a preliminary phase before Algorithm 5 begins. This disjoint core phase is shown in Algorithm 6. Experiments in Section 3.7 demonstrate that the disjoint core phase is very beneficial. It quickly increases the known lower bound, which may be an important factor for some applications. Changing the order in which cores are found also means that the overall time spent solving MCHS instances is reduced, since it allows trivial MCHS instances to be identified. The disjoint core phase can also decrease the total number of cores required. In some rare cases, the maxsat instance is even solved based on the initial set of disjoint cores alone.

Algorithm 6: The disjoint core phase.

```
   /* F is a MAXSAT instance.                                                    */
   /* Returns a set of disjoint cores of F and a minimum cost hitting set for them.*/
 1 DisjointCores (F)
 2 K = {}
 3 hs = {}
 4 while true do
 5     (sat?,κ) = SatSolver(F)
 6     if sat? then
 7         break
 8     else
 9         K = K ⋃ minimize(κ)
10         c_min = argmin_{c∈κ}{wt(c)}
11         hs = hs ⋃ {c_min}
12         F = F \ κ
13 return (K, hs)
```

3.6.4 Other Methods

In addition to minimizing the cores and performing a preliminary disjoint core phase, a few other simple techniques can be used that may increase the diversity of cores. As mentioned in Section 3.5, the state of MINISAT persists from one SAT solving episode to the next during the execution of MAXHS. However, this may adversely affect the diversity of cores that the SAT solver produces. In order to combat this effect we experimented with deleting the learnt clauses between iterations, and modifying the variable ordering heuristic by either increasing the frequency of random decisions or inverting the variables' activities so that variables with high activity from the previous SAT episode are given low activity and vice versa. The effectiveness of these techniques in practise is reported in Section 3.7.

A possible direction for future work is to adapt existing methods that find minimal cores with specific properties (Ryvchin and Strichman, 2011) to find cores that are likely to increase the cost of the MCHS.

3.7 Experimental Evaluation

3.7.1 Experimental Setup

In this section we describe the set of maxsat instances that were used to evaluate solver performance, the computing environment in which all experiments were conducted, and the resource limits we imposed on the solvers.

Choice of maxsat Instances The performance of maxhs was evaluated on all instances appearing in the 2006-2012 maxsat Evaluations except those in the Random category. This is the first experimental study that evaluates state-of-the-art maxsat solvers on such a large and comprehensive collection of non-random instances. The Evaluation instances have played an important role in the development of all existing state-of-the-art maxsat solvers. However, it is conventional in the literature to use a much smaller subset of these instances to evaluation new maxsat solvers. The different subsets used vary greatly by both size and their restriction to particular sub-categories, even though almost all maxsat algorithms are meant to be robust and general-purpose. Therefore, the best existing comparison of maxsat solvers can be found in the maxsat Evaluations, where all solvers are tested on the same set of instances. However, in the most recent two maxsat Evaluations (2011 and 2012), the memory limit for the solvers was less than 0.5GB (Argelich et al., 2007–2012). Given that some of the maxsat instances themselves are close to this size, the tight memory limit probably affected the fair comparison of different solvers. Furthermore, the state-of-the-art solver bincd did not participate in the maxsat Evaluations. Our experimental study goes a step farther, by including all instances used in any year of the maxsat Evaluations, using a more modern computing environment, and including solvers that were never entered in the Evaluations.

The Evaluation instances represent the largest and most diverse set of maxsat instances that is currently publicly available. However, the Evaluation instances are also

inevitably biased because they are only a small finite set, accumulated under specific circumstances. The natural bias is to include instances that are solvable by existing MAXSAT solvers under reasonable resource limits. Almost by definition, if a potential MAXSAT application is too difficult for current MAXSAT solvers, then *it is not a* MAXSAT *application*. In future work, we would like to investigate new applications for MAXSAT. We believe that our MAXHS approach will be able to adapt to a variety of future applications because of its flexibility and power as a hybrid approach. However, for now, it is important that we have included as many existing publicly available MAXSAT instances as possible.

Many MAXSAT instances were used in more than one year of the 2006-2012 MAXSAT Evaluations. In most cases, it is easy to identify such duplicated instances by hand, according to their filenames or other clues. If an instance appeared in more than one year of the Evaluations, only its first occurrence was kept. After the duplicates were removed, a set of 4502 instances remained.

Computing Environment All experiments reported in this thesis were conducted using the facilities of SHARCNET (the Shared Hierarchical Academic Research Computing Network), a High Performance Computing Consortium within Compute Canada (SHARCNET, 2013). Experiments with the SAT4J solver were run on the "guppy" cluster, on Intel Xeon 2.8GHz processors and the CentOS 5.5 operating system. All other experiments were conducted on the "redfin" cluster, on AMD Opteron 6172 2.1GHz processors and the CentOS 6.3 operating system.[4]

Resource Limits The time limit for each test was 1200 seconds. The memory limit was set to 2.5GB, and this amount of memory was explicitly reserved for each process.

[4]We discovered after running a large number of experiments on the "redfin" cluster that SAT4J could not be run on the "redfin" cluster for technical reasons.

Competing Solvers We ran all experiments with the following MAXSAT solvers: WPM1 (with the latest 2012 improvements (Ansótegui et al., 2012)), WPM2 (versions 1 and 2 (Ansótegui et al., 2010)), BINCD (msuncore-2011606-linux64 with the option "-r bin-core-dis" (Heras et al., 2011)), WBO (Manquinho et al., 2009), MINIMAXSAT (Heras et al., 2008), SAT4J (version 2.3.0 (Berre and Parrain, 2010)), and AKMAXSAT_LS (version 1.1 (Kügel, 2010)).

All of these solvers are able to solve MAXSAT in its most general form, i.e., weighted partial MAXSAT, and thus have the widest range of applicability. This set of solvers includes recently developed solvers utilizing the sequence of SAT approach (BINCD, WPM1, WPM2), some older ones (SAT4J and WBO), and two prominent Branch and Bound based solvers (AKMAXSAT_LS and MINIMAXSAT).

Several other MAXSAT solvers were entered in the recent MAXSAT Evaluations. These include QMAXSAT (Koshimura et al., 2012) and PM2 (Ansótegui et al., 2009),[5] which use the sequence of SAT approach but are restricted to instances with uniform weight soft clauses (i.e. partial MAXSAT). We omit these solvers from our experimental evaluation because they are specialized to a small subset of the instances we are interested in solving. We did not experiment with PWBO (Martins, Manquinho, and Lynce, 2012a) because it is a parallel MAXSAT solver, while all other solvers use serial algorithms.

For the case of Branch and Bound solvers, we limited our experiments to two solvers (MINIMAXSAT and AKMAXSAT_LS) to save time in running the experimental study. The other Branch and Bound solvers (i.e. IUT_RR (Ramezani and Mousavi, 2012), WMAXSATZ-2009 (Li et al., 2009), INCMAXSATZ (Lin et al., 2008) and WMAXSATZ+ (Li et al., 2007, 2010)) use lower bound techniques that are quite similar to AKMAXSAT_LS,[6] but AKMAXSAT_LS solved more problems in the Random and Crafted categories of the 2012 MAXSAT Evaluation so we chose it as a representative. We included MINIMAXSAT in our experiments even though it did not participate in the recent MAXSAT Evaluations

[5] See Section 2.5.2 for a description of these solvers.
[6] See the discussion in Section 2.5.1.

because it uses a distinct lower bound technique. Our experimental results show that the performance of MINIMAXSAT is still comparable to the state-of-the-art.

We also solved the MAXSAT instances with the MIP solver CPLEX (version 12.2) after applying the following simple encoding of MAXSAT to MIP.[7]. First, the clauses of the b-variable relaxation $\mathcal{F}^b$ are encoded as linear inequalities, using the standard method where a clause c is converted to the linear inequality $\sum_{j:p_j \in c} p_j + \sum_{i:\neg p_i \in c}(1-p_i) \geq 1$. Note that a negative literal p is encoded as $(1 - var(p))$ and a positive literal is encoded as $var(p)$. For example, the clause $(x, y, \neg z, b_1)$ becomes the linear inequality $x+y+(1-z)+b_1 \geq 1$. Second, the objective function becomes minimizing $\sum_i wt(c_i) \times b_i$. The MIP thus tries to find a setting for the propositional variables that satisfies all of the clauses and has minimum *bcost*.

Versions of MAXHS We compare eight versions of MAXHS. All of these versions build on the "basic" version of the original algorithm as shown in Algorithm 5, with zero or more of the techniques described in Section 3.5 added. The eight versions are specified in Table 3.1.

MAXHS Version	Minimal Cores	Disjoint Phase	Invert Activity	Delete Learnts	Re-refute Cores
MAXHS					
MAXHS-min	✓				
MAXHS-disj		✓			
MAXHS-min-disj	✓	✓			
MAXHS-min-disj-inv	✓	✓	✓		
MAXHS-min-disj-del	✓	✓		✓	
MAXHS-min-disj-inv-del	✓	✓	✓	✓	
MAXHS-reref-disj-inv-del		✓	✓	✓	✓

Table 3.1: The eight versions of MAXHS that we evaluate in this chapter.

[7]The origins of this encoding are not clear. However, it is already known to most people in the field.

3.7.2 Experimental Results

We first evaluate the performance of the different versions of MAXHS and the competing solvers in terms of the number of problems they are able to solve within the time limit. Then, we focus on understanding the practical behaviour of MAXHS in terms of the trade-off between the time taken by the SAT solver and CPLEX.

Overall Performance of MAXHS

The overall performance of the different versions of MAXHS and the competing solvers is shown in a cactus plot in Figure 3.1. We observe that each of the techniques we proposed in Section 3.5 improves the total number of problems solved by the MAXHS approach. The greatest gains in overall performance are due to the disjoint core phase and especially minimal cores. The tweaks to MINISAT of inverting the activities and deleting learnt clauses, when used together, lead to significant gains as well. This can be seen by comparing MAXHS-min-disj and MAXHS-min-disj-inv-del which solve 2891 and 2996 instances respectively. Finally, minimal cores are much superior to reducing cores through re-refutation only (see MAXHS-reref-disj-inv-del vs. MAXHS-min-disj-inv-del).

Figure 3.1 also shows how MAXHS compares to existing MAXSAT solvers. The basic version of MAXHS is not competitive with existing state-of-the-art solvers. However, the improvements we proposed are very effective and the resulting version, MAXHS-min-disj-inv-del, is well ahead of sophisticated MAXSAT solvers including AKMAXSAT_LS, WBO and WPM2. Three competing MAXSAT solvers, WPM1, BINCD and MINIMAXSAT still outperform our solver. However, all MAXSAT solvers are outperformed by CPLEX.

The good performance of CPLEX in comparison to MAXSAT solvers on MAXSAT instances is surprising since MAXSAT solvers are specialized to this domain. This observation motivates us to find ways to further exploit CPLEX within the MAXHS approach.

Tables 3.2 and 3.3 show the overall results broken down by benchmark family. The tables divide the instances according to whether they appear in the Crafted or Industrial

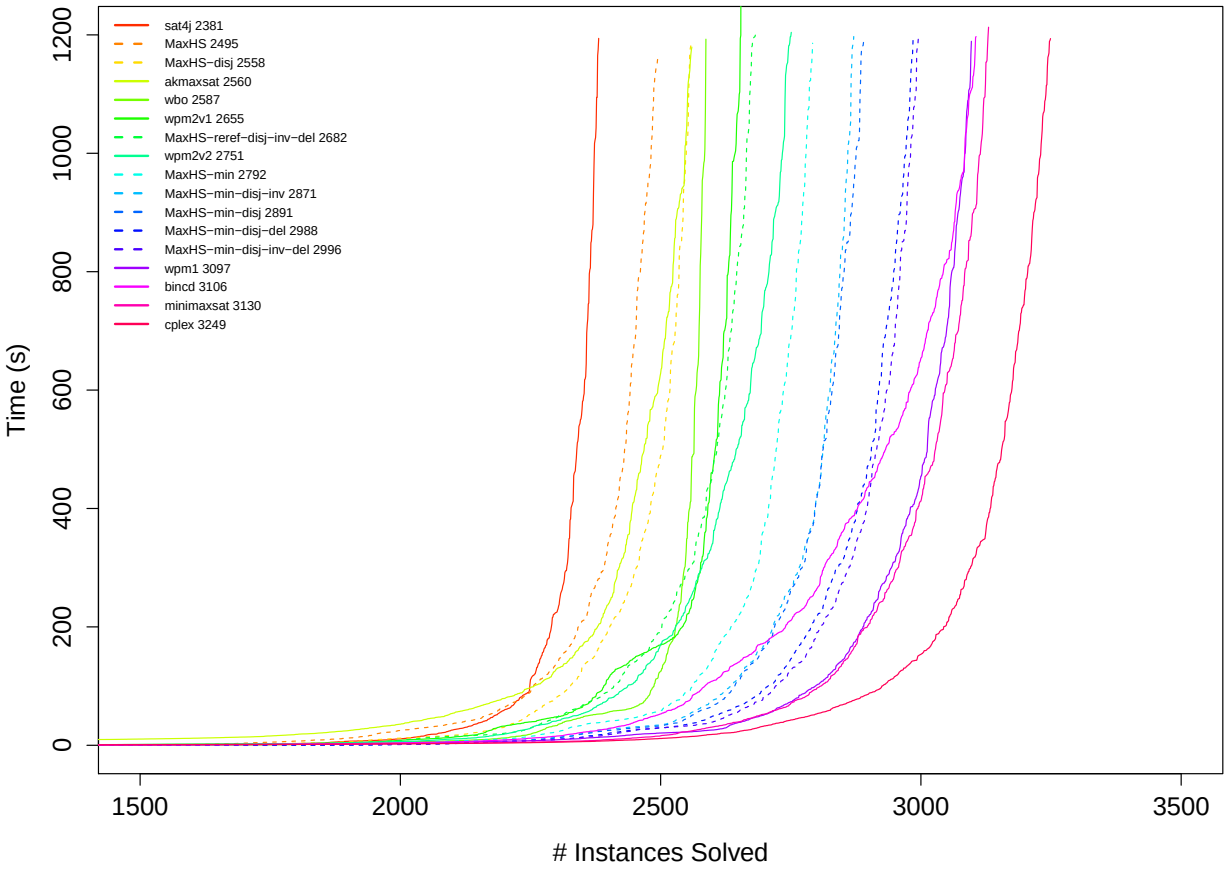


Figure 3.1: Runtime results for the competing solvers and eight versions of MaxHS. Shows how many problems were solved within each time limit. The total number of instances solved is given in the legend after the solver's name.

categories of the MaxSAT Evaluations. An important observation is that CPLEX does very well on the Crafted instances, but is not among the top solvers on Industrial instances. However, our MaxHS approach is very competitive on the Industrial instances, where it is the second-best solver overall. This suggests that we should be able to take better advantage of CPLEX in order to improve the performance of MaxHS on Crafted instances, resulting in a very robust solver for MaxSAT.

Trade-offs in MaxHS

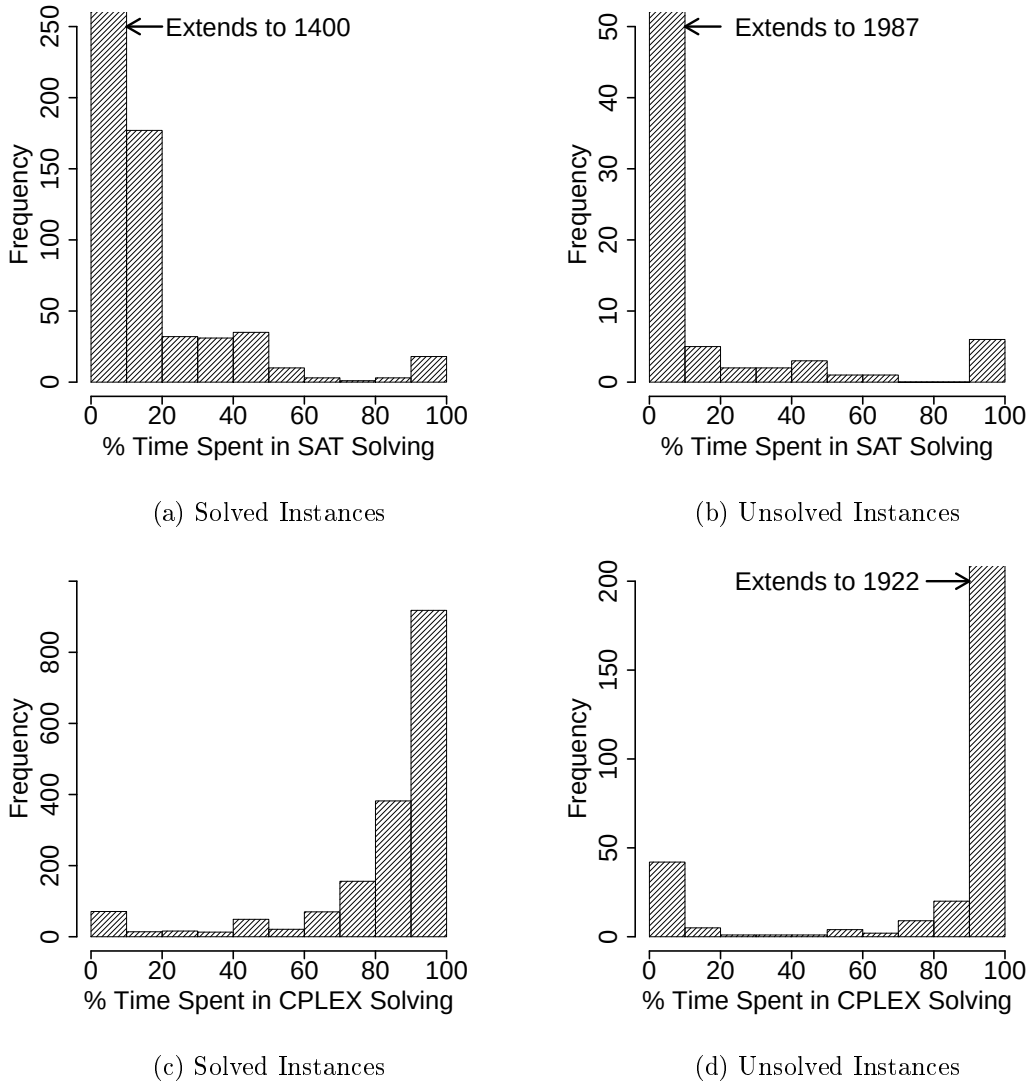
Next, we focus on understanding the practical behaviour of MaxHS and the techniques from Section 3.5. Section 3.4 described how the MaxHS approach is affected by three

main factors: how hard it is to find the cores, how difficult it is to solve the hitting set problems, and how many cores are required in total. Which of these factors is responsible for limiting the performance of maxhs in practise? How are these factors affected by the various techniques like core minimization?

To answer these questions, we collected some statistics from each run of our maxhs solver. We can think of cplex and minisat as two subroutines invoked by our algorithm, and ask how much time is spent in each of these subroutines.[8] For every run of maxhs we recorded the number of calls to cplex, the total time spent in cplex's solve routine, and the total time spent in minisat.

We begin by examining the percentage of the total runtime that was spent in SAT solving and in calls to cplex for the basic version of maxhs. Figures 3.2a and 3.2b show histograms for the percentage of total runtime spent in SAT solving. In Figure 3.2a, only the instances that the basic version of maxhs solved within the resource limits are included. All instances that maxhs failed to solve are included in Figure 3.2b. Figures 3.2c and 3.2d show the corresponding histograms for the percentage of total runtime spent in calls to cplex. We observe that on most instances (solved or unsolved), the SAT solving time is a small percentage of the total runtime, while the time spent in cplex calls is a very high percentage of total runtime. This explains why using more SAT calls in order to minimize the cores pays off. These observations motivate us to focus on reducing the time spent in solving the MIP subproblems, even if it increases the work done by the SAT solver.

[8] Note that the cplex and minisat times do not add up to the total runtime of maxhs, which also includes the time to parse the input and communicate between cplex and minisat.



(a) Solved Instances

(b) Unsolved Instances

(c) Solved Instances

(d) Unsolved Instances

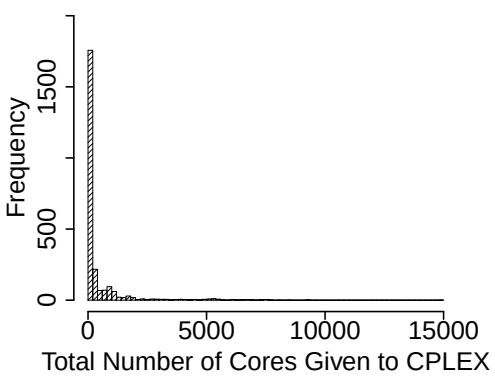
Figure 3.2: Histograms for the percentage of runtime spent in SAT solving and in calls to cplex, for the basic version of maxhs.

The third property of maxhs mentioned in Section 3.4, is the total number of cores it requires to find the maxsat solution. The number of cores given to cplex is shown in the histograms of Figure 3.3 for the basic version of maxhs, on solved and unsolved instances. We observe that on instances that maxhs is able to solve, usually the number of cores is smaller. Many instances (621, see Figure 3.3b) can be solved without any cores at all. This can occur, for example, if the maxsat instance is in fact satisfiable, or if the

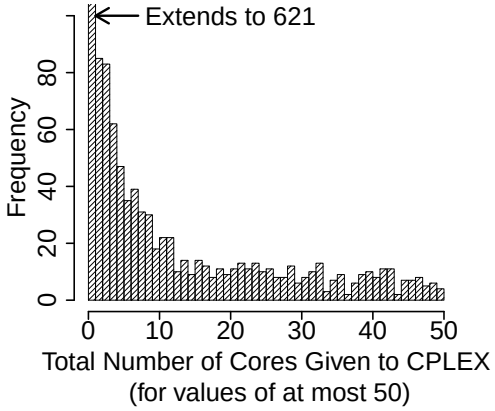
hard constraints imply via unit propagation that some set of soft clauses is falsified which is costly enough to exceed the upper bound. However, there are many instances that find hundreds or thousands of cores and are still solvable, as can be seen in Figure 3.3c. Yet only very few solved instances required more than 5000 cores.

On the instances that maxhs failed to solve within the timeout, the number of cores found before timeout tends to be large, as can be seen in Figure 3.3d.[9] This suggests that the number of iterations performed by maxhs is the reason the algorithm failed to solve these instances, rather than the difficulty of SAT solving or of solving the MCHS problems. Some of these instance might be solvable if maxhs can find a different, smaller, set of cores that is sufficient to prove the maxsat solution. However, we know from the examples in Section 3.4 that some maxsat instances do not have such a small set of cores. In such cases it is important to reduce the overhead of SAT solving and calls to cplex in order to allow as many cores as possible to be found.

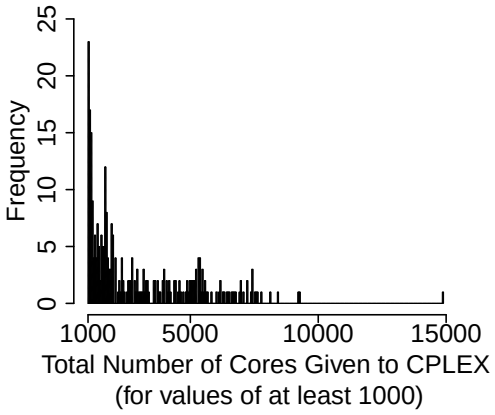
[9] The unsolved instances for which the number of cores is zero are cases where the SAT solver consumed all of the memory or time limit during the initial refutation of the maxsat instance.



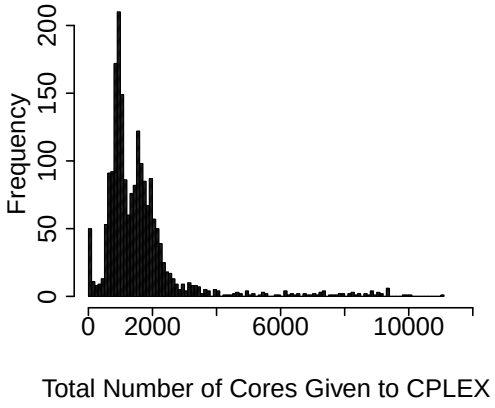
(a) Solved Instances



(b) Solved Instances



(c) Solved Instances



(d) Unsolved Instances

Figure 3.3: These histograms report the number of cores given to CPLEX by the basic version of MAXHS.

We also compare the behaviour of different versions of MAXHS, to investigate why the various proposed techniques pay off. The final value of the lower bound (i.e. the cost of the last computed MCHS) indicates how much progress MAXHS has made towards the solution, even if the time limit is reached before the MAXSAT solution is found. So we can measure the progress made by each CPLEX call, by dividing the final LB by the number

of cplex calls, to get the average increment to the LB per cplex call. The average time spent by a call to cplex is obtained by dividing the total cplex time by the number of calls to cplex. And by dividing the total SAT time by the number of calls to cplex, we get an idea of how much time is spent finding cores for each call to cplex.

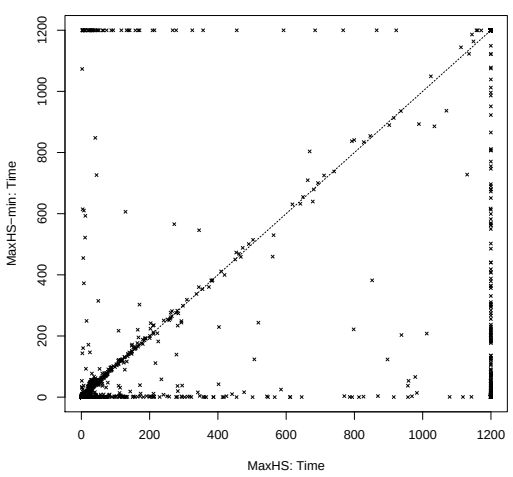
The following set of graphs (Figures 3.4-3.7) compares versions of maxhs according to four dimensions: the total runtime of the solver, the LB increment per cplex call, the SAT time per cplex call, and the cplex time per cplex call. All 4502 instances appear in each of these graphs. Note that since the data points often overlap each other it sometimes appears that fewer instances are included in a graph (e.g. see Figure 3.4c). In each graph, the less sophisticated version of maxhs is on the horizontal axis, and the more sophisticated version of maxhs is on the vertical axis.

We first investigate the effect of using minimal cores by comparing the basic version of maxhs with maxhs-min in Figure 3.4. Recall from Figure 3.1 that the overall performance of maxhs-min is significantly better than maxhs. It is clear from Figure 3.4a that minimal cores often improves the runtime by orders of magnitude. This improvement in performance is explained by Figures 3.4b and 3.4d. Using minimal cores has a consistent effect of increasing the increment in the lower bound per cplex call. Moreover, the time required by cplex to solve each hitting set problem generally decreases. Thus, by spending more time in the SAT solver to minimize the cores, we are able to give better cores to cplex, which both reduce the difficulty of the hitting set instances and increase the progress made by each cplex call.

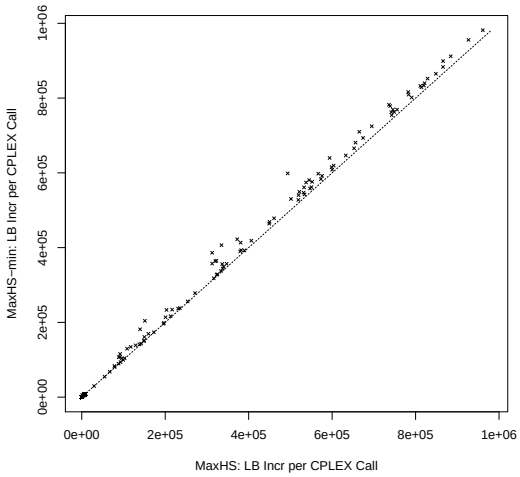
However, one some instances the runtime is instead increased by using minimal cores. This can be partly explained by the fact that the time spent in SAT solving sometimes increases exponentially, as shown in Figure 3.4c. In the basic version of maxhs, the SAT time per cplex call is always close to zero. In a small number of cases, minimizing the cores takes a very long time, consuming all of the 1200 second time limit.

The effect of the disjoint cores phase is shown in Figure 3.5. The benefit of the

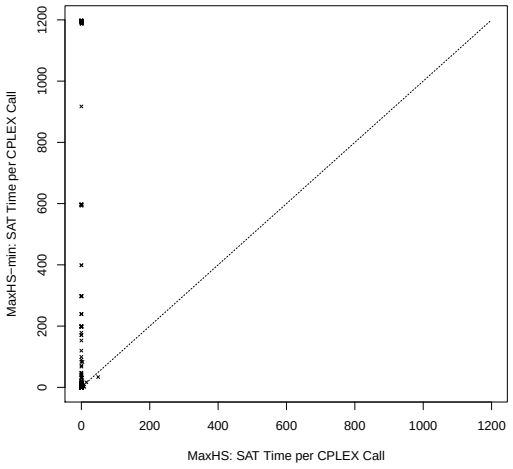
disjoint core phase is that it greatly increases the amount the lower bound is incremented per cplex call. This is exactly as we expect, since during the disjoint core phase the lower bound is always increased without requiring calls to cplex. The total runtime is decreased because fewer calls to cplex are required. The disjoint core phase does not significantly effect the time spent in SAT solving, except in a very small number of cases where it can make the cores more difficult to find. The disjoint cores phase generally reduces the cplex time per cplex call. maxhs uses the incremental nature of cplex, meaning that cplex uses information from previous solving episodes to solve the current hitting set problem. So a possible explanation for the reduction in cplex time per call is that cplex benefits from being initialized with the set of disjoint cores. From Figure 3.6, we see that none of these statistics provides a convincing explanation for the benefit of using the two tweaks to minisat (inverting the activities and deleting learnt clauses). Finally, we compare the basic version of maxhs with the best performing version, maxhs-min-disj-inv-del. By comparing the graphs in Figure 3.7 to the previously discussed graphs, we can see that the behaviour of maxhs-min-disj-inv-del appears to be a superposition of the behaviour of each of the techniques. In particular, by spending more time in the SAT solver to minimize the cores and find as many disjoint cores as possible, the effectiveness of the information provided to cplex is generally improved. The net effect is a significant improvement in total runtime.



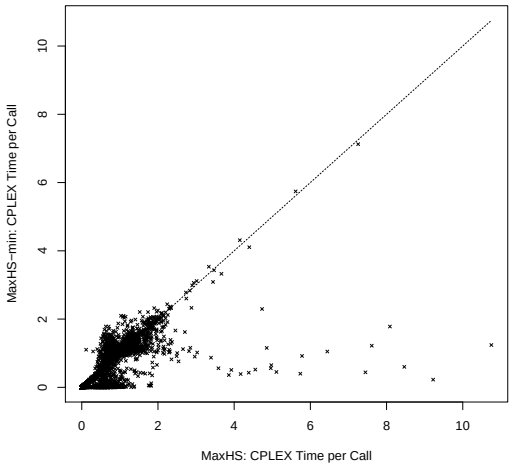
(a) Runtime



(b) LB Increment per CPLEX Call

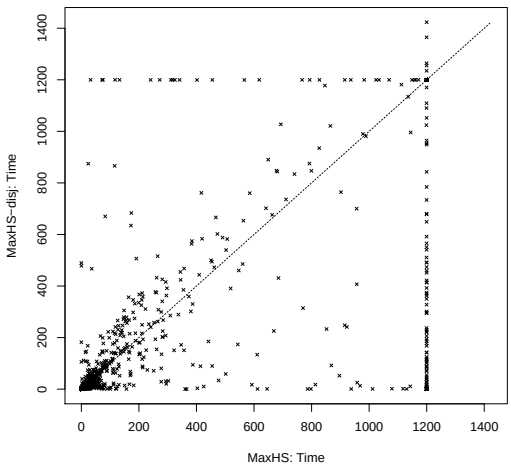


(c) SAT Time per CPLEX Call

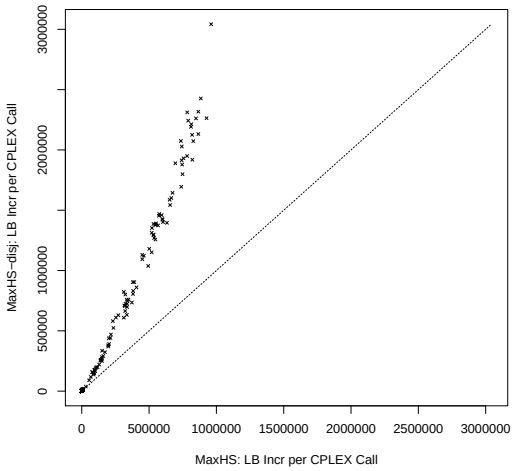


(d) CPLEX Time per CPLEX Call

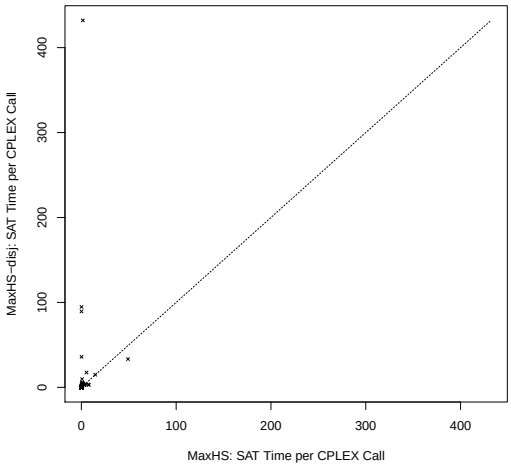
Figure 3.4: MaxHS vs. MaxHS-min



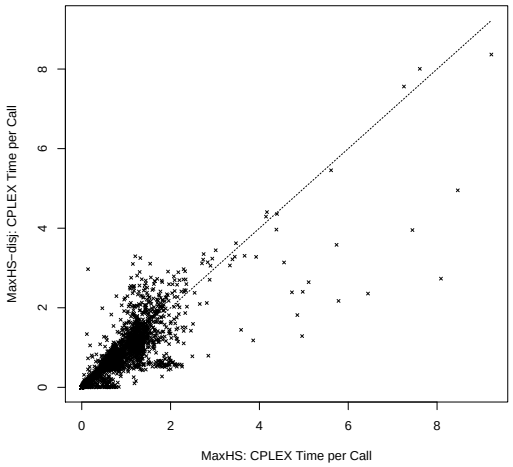
(a) Runtime



(b) LB Increment per CPLEX Call

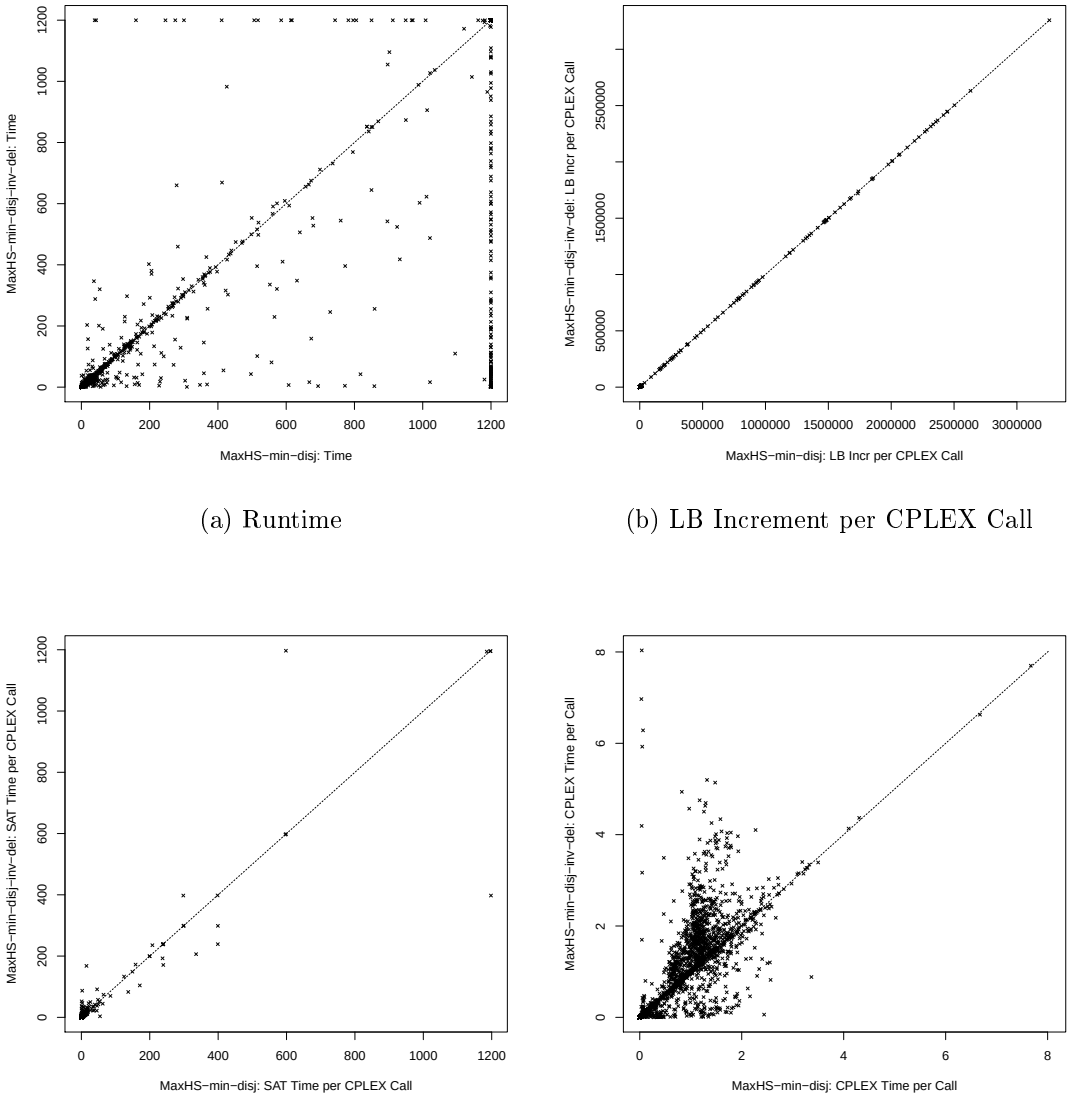


(c) SAT Time per CPLEX Call



(d) CPLEX Time per CPLEX Call

Figure 3.5: MaxHS vs. MaxHS-disj



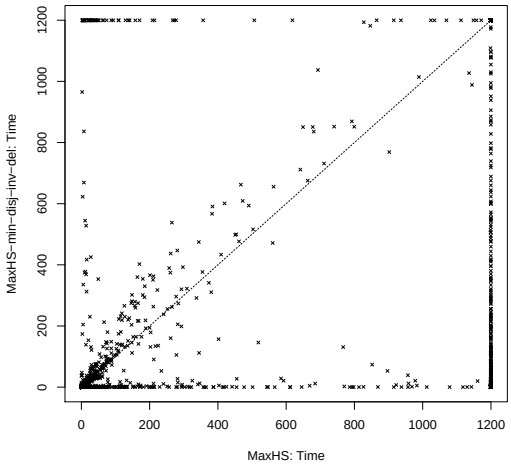
(a) Runtime

(b) LB Increment per CPLEX Call

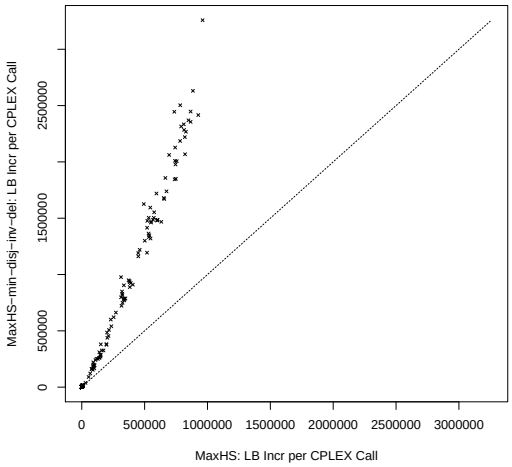
(c) SAT Time per CPLEX Call

(d) CPLEX Time per CPLEX Call

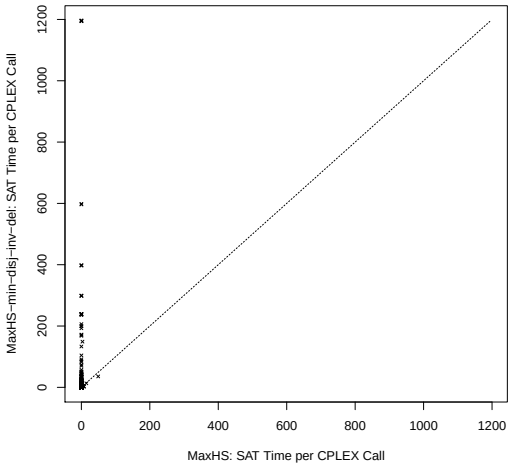
Figure 3.6: MaxHS-min-disj vs. MaxHS-min-disj-inv-del



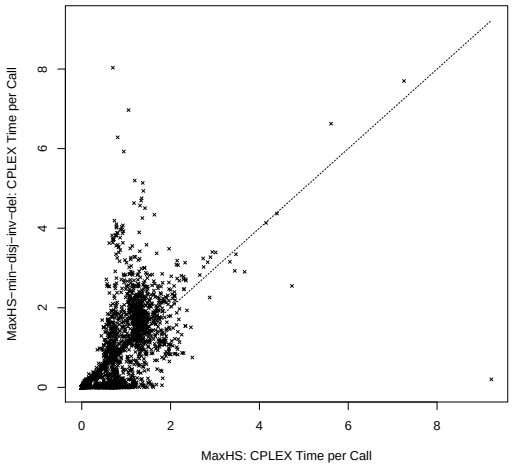
(a) Runtime



(b) LB Increment per CPLEX Call



(c) SAT Time per CPLEX Call



(d) CPLEX Time per CPLEX Call

Figure 3.7: MaxHS vs. MaxHS-min-disj-inv-del

3.8 Related Work

In existing maxsat solvers that pose a series of decision problems, the difficulty of solving the decision problems is the main factor that limits performance. This is the case whether the decision problems are translated to CNF so that they can be solved by a SAT solver, or a Pseudo-boolean solver is used to solve them natively. In contrast, the maxhs approach involves posing a sequence of decision problems that are no more difficult than refuting the original maxsat formula. maxhs does not add any cardinality constraints to the SAT instance, and weighted clauses never get duplicated as in wpm1 and wbo. Instead, the arithmetic constraints specifying that at least one clause from every core needs to be falsified are dealt with directly by the minimum hitting set solver.

The maxhs approach is closely related to the Implicit Hitting Set (IHS) problem as described in (Karp, 2010; Chandrasekaran, Karp, Moreno-Centeno, and Vempala, 2011a). In IHS problems one is trying to compute a minimum cost hitting set without knowing ahead of time the collection of sets that need to be hit. Instead, one is provided with an oracle that when given the current candidate hitting set, either declares the candidate to be a correct hitting set or returns a new un-hit set from the implicit collection. In the maxhs algorithm, the cores of $\mathcal{F}$ form the collection of sets to be hit, cplex computes candidate hitting sets, and the sat assumption test acts as the oracle deciding if the current candidate hitting set is correct, returning a new un-hit core if it is not. However, the sat assumption test may take exponential time, while the oracle in IHS is assumed to run in polynomial time.

It would be interesting to encode IHS problems as maxsat instances and try solving them with the MaxHS solver. One advantage to the MaxHS approach is that it does not require a special-purpose polynomial time oracle to be created for each application. Rather, the task would be to devise a maxsat encoding of the application, which is conceptually simpler.

The fact that a MCHS of a set of cores is a lower bound on the maxsat solution

follows from the well-known duality between minimal unsatisfiable sets (MUSes) and maximal correction sets (MCSes) (Bailey and Stuckey, 2005).

3.9 Conclusion

This chapter introduced the maxhs approach for solving maxsat. The maxhs algorithm decouples the satisfaction and optimization components of solving maxsat, by using a SAT solver to find cores and a MIP solver like cplex to solve minimum cost hitting set problems over those cores. The maxhs approach allows the differing strengths of these two types of solvers to be effectively exploited. The result is a maxsat solver, maxhs-min-disj-inv-del, that is comparable in performance to existing state-of-the-art maxsat solvers.

Additionally, we were surprised to discover that the MIP solver cplex is a very effective maxsat solver, especially on Crafted instances. Therefore, in the following chapters we investigate two approaches to improving maxhs based on adding more information to the cplex problems.

Family	#	wpm2 v1	wbo	sat4j	wpm2 v2	bincd	wpm1	ak	cplex	mini	MaxHS disj	MaxHS	MaxHS reref disj inv del	MaxHS min	MaxHS min disj inv	MaxHS min disj	MaxHS min disj del	MaxHS min disj inv del
ms/spinglass/	20	1	3	0	1	0	0	20	19	20	0	0	0	0	0	0	0	0
wms/kexu/frb-wcnf/	35	10	5	3	5	9	5	15	20	15	10	11	10	10	10	10	10	10
pms/csp/sparseloose	20	20	19	17	20	20	20	20	20	20	9	9	19	9	11	11	20	20
wpms/pb/factor/	186	186	172	174	186	186	168	186	186	186	186	186	186	186	186	186	186	186
pms/csp/denseloose	20	19	5	7	18	20	16	20	20	20	0	0	5	0	0	1	5	10
KnotPipatsrisawat	350	0	173	5	0	0	161	119	245	117	12	12	57	1	9	11	40	61
pms/queens	7	7	5	6	7	7	7	7	7	7	2	2	2	3	3	4	5	5
wpms/aucregions	84	0	0	3	37	6	0	84	84	84	34	39	34	39	33	35	34	35
ms/cut/spinglass	5	1	2	0	1	1	1	3	3	3	0	0	0	0	0	0	0	1
pms/jobshop	4	0	2	3	4	4	3	0	0	2	4	4	4	4	4	4	4	4
wpms/planning	71	48	47	67	53	65	64	52	70	71	50	49	46	68	68	69	69	69
pms/maxone/struc	60	25	1	57	39	59	30	13	52	60	16	12	5	46	45	45	45	46
ms/ramsey	48	34	34	32	34	34	34	35	34	35	34	34	34	34	34	34	34	34
pms/clique/rand	96	62	0	58	64	67	0	96	96	96	4	4	4	4	4	4	4	4
wms/cut/spinglass	5	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	1	1
wpms/pb/miplib	16	6	2	8	6	7	6	4	6	5	5	5	6	7	7	7	7	7
wms/ramsey	48	35	34	29	35	36	34	37	36	37	34	34	34	34	34	34	34	34
ms/cut/dimacs	62	4	4	2	4	6	5	52	20	48	3	3	4	3	4	3	4	4
wpms/aucsched	84	3	0	56	84	66	84	77	84	84	82	83	82	83	82	82	82	81
ms/bip-cut-140-630	100	0	0	0	0	0	0	100	0	83	0	0	0	0	0	0	0	0
wpms/min-enc/warehouses	18	0	4	2	1	1	14	2	18	2	1	1	1	1	1	1	1	1
pms/min-enc/kbtree	54	0	12	12	14	15	14	18	54	22	8	10	11	11	11	11	11	12
wpms/aucpaths	88	0	0	3	6	0	52	88	88	88	88	88	88	88	88	88	88	88
pms/csp/sparsetight	20	2	14	0	3	20	20	20	20	20	0	0	0	0	0	0	0	0
wpms/spot5log	21	8	6	2	9	11	12	4	6	4	6	6	6	6	6	6	6	6
pms/maxone/3sat	80	34	33	51	34	80	71	80	80	80	20	19	20	24	25	25	25	25
wms/cut/rand	40	0	0	0	0	0	0	40	12	40	0	0	0	0	0	0	0	0
wpms/QCP	25	25	25	16	25	25	25	20	25	20	25	25	25	25	25	25	25	25
pms/clique/struc	62	19	10	13	17	18	8	36	32	36	10	10	10	10	10	10	10	10
wms/cut/dimacs	62	3	4	1	3	4	5	60	22	55	3	3	3	3	3	3	3	3
ms/cut/rand	40	0	0	0	0	0	0	40	4	40	0	0	0	0	0	0	0	0
wpms/min-enc/planning	56	7	30	51	43	53	52	35	51	56	39	37	31	54	54	54	54	54
pms/frb	25	6	0	0	5	0	0	5	9	5	0	0	0	0	0	0	0	0
wpms/spot5dir	21	9	5	1	9	11	10	4	17	3	6	6	6	6	6	6	6	6
pms/csp/densetight	20	0	13	0	1	19	19	20	20	20	0	0	0	0	0	0	0	0
pms/pb/garden	7	5	5	4	5	5	5	5	6	5	5	5	5	5	5	5	5	5
Total	1960	579	669	683	773	855	945	1421	1470	1493	696	697	738	764	768	774	818	847

Table 3.2: Crafted instances: for each of the competing solvers and eight versions of MaxHS, this table shows the number of instances solved in each benchmark family. For each family, the number of instances in the family is shown in column '#'. The name of the family begins with either 'ms', 'pms', 'wms' or 'wpms' which indicates whether or not the instances contain hard clauses ('p') and whether or not their soft clauses have non-uniform weights ('w'). The solvers are ordered by the total number of problems they solve.

Family	#	ak	mini	sat4j	cplex	wbo	wpm2 v2	wpm2 v1	wpm1	bincd	MaxHS	MaxHS disj	MaxHS reref disj inv del	MaxHS min	MaxHS min disj inv	MaxHS min disj	MaxHS min disj inv del	MaxHS min disj del
haplo-ped	100	0	0	15	9	73	21	45	91	23	20	21	27	38	46	55	46	54
pb-nencdr	128	0	64	78	23	63	74	59	69	116	31	31	48	72	98	97	109	111
pms/bcp-mtg	215	149	208	183	193	172	215	215	215	215	138	157	150	186	196	198	214	214
wpms/up-u98	80	0	0	67	80	80	64	80	80	79	80	80	80	80	80	80	80	80
ms/Safar	112	4	3	6	19	69	57	75	88	71	72	77	75	35	34	34	34	35
pms/pb/primes	86	71	76	60	78	23	67	63	46	76	61	65	59	73	74	74	74	74
pms/bcp-syn	74	31	27	12	71	32	33	34	40	45	60	63	65	67	67	67	67	67
pms/circtracecomp	4	0	1	0	0	0	4	3	3	4	0	0	0	0	0	0	0	0
pms/hap-asmbly	6	0	0	0	2	5	5	5	4	0	2	5	5	5	5	5	5	5
pb-nlogencdr	128	1	103	70	24	67	101	67	88	128	34	37	78	81	99	99	118	118
pms/bcp-fir	59	12	13	10	58	41	49	48	53	55	13	15	16	17	18	18	18	18
pms/pbo-rout	15	0	14	14	14	15	15	15	15	15	0	9	10	7	15	15	15	15
pms/pseudo/rout	15	0	14	15	15	15	15	15	15	15	0	7	7	7	15	15	15	15
aes	7	1	1	0	2	0	0	0	0	1	1	1	1	1	1	1	1	1
timetabling	32	0	0	1	0	9	13	14	13	12	10	9	9	11	6	10	6	12
pms/bcp-hipp	1183	772	982	1080	962	1114	1155	1153	1154	1164	1085	1090	1125	1144	1143	1143	1140	1144
pms/pb/logic-syn	17	2	2	1	16	5	5	5	7	7	14	14	16	16	16	16	16	16
ms/circdebug	9	0	0	0	1	7	5	8	9	7	9	8	9	4	4	4	4	4
upgrade	100	0	0	3	100	100	0	99	100	97	100	100	100	100	100	100	100	100
pms/protein-ins	12	2	11	3	1	1	2	3	1	2	1	1	1	1	2	2	2	2
wpms/protein-ins	12	2	10	3	1	1	2	3	1	2	1	1	1	1	1	1	2	2
pms/bcp-msp	148	92	108	77	110	26	76	67	60	117	66	71	62	82	83	83	83	83
Total	2542	1139	1637	1698	1779	1918	1978	2076	2152	2251	1798	1862	1944	2028	2103	2117	2149	2170

Table 3.3: Industrial instances: for each of the competing solvers and eight versions of MaxHS, this table shows the number of instances solved in each benchmark family. For each family, the number of instances in the family is shown in column '#'. The name of the family begins with either 'ms', 'pms', 'wms' or 'wpms' which indicates whether or not the instances contain hard clauses ('p') and whether or not their soft clauses have non-uniform weights ('w'). The solvers are ordered by the total number of problems they solve.

Chapter 4

Constraining the Hitting Sets

Chapter 3 introduced the MAXHS algorithm, which decomposes the MAXSAT problem into a series of SAT problems and hitting set problems. The hitting set problems are solved by expressing them as integer programs and applying a general MIP solver. Neither the SAT solver nor the hitting set problem alone has enough information to solve the entire MAXSAT problem. The SAT solver does not have any information about the clause weights, and the MIP solver knows nothing about the original variables and clauses. In this chapter we consider two ways of strengthening the information available to the MIP solver. The main observation is that there are additional constraints that the hitting sets must satisfy in order to really correspond to the MAXSAT solution. By enforcing these additional constraints on the hitting set, many minimal hitting sets can be ruled out without requiring the SAT solver to find cores to rule them out. It is hoped that constraining the hitting set problems will both make the MCHS problems easier to solve, and reduce the total number of iterations.

In Section 4.1, we define a *realizability* condition that the hitting sets must satisfy. However, the MIP solver is ill-suited to enforcing the realizability condition. Then, in Section 4.2, we show how the b-variable relaxation can be extended with *equivalences* so as to allow the SAT solver to produce general clausal constraints, rather than just cores,

over the b-variables. We find that this is a more promising approach to constraining the hitting set problems, because it takes better advantage of the MIP solver.

4.1 Realizable Hitting Sets

Any complete truth assignment partitions the soft clauses of a MAXSAT instance into two sets: those that it falsifies and those that it satisfies. The goal of solving the MAXSAT instance is to find the truth assignment that minimizes one side of this partition: the set of falsified soft clauses. The hitting sets returned by the MIP solver can be thought of as proposing candidate partitions: the clauses in the hitting set should be falsified, and the rest should be satisfied. However, the MIP solver is working with incomplete information, so the partition it proposes may not be achievable by any truth assignment. For example, if two clauses in the hitting set hs turn out to contain conflicting literals, no truth assignment will be able to falsify every clause in hs, and therefore this hitting set can not correspond to a MAXSAT solution. However, the MIP solver in Algorithm 5 (page 41) does not know about the literals contained in each clause, so it can return hitting sets with this flaw.

This section shows how more information can be added to the hitting set problem, in order to prevent hitting sets being generated and obviously can't correspond to a MAXSAT solution. This allows some hitting sets to be ruled out without requiring a core to be produced to rule them out. Experiments show that this can greatly reduce the total number of cores needed to solve the MAXSAT instance, and also significantly reduce the total runtime of the MAXSAT solver.

The hitting set that causes MAXHS to terminate will have the property that it is possible to falsify every clause in it, while also satisfying all of the hard clauses. A hitting set that meets these two requirements is called *realizable*.

Definition 16 (Realizable)**.** *A hitting set hs is realizable in a* MAXSAT *problem $\mathcal{F}$ if*

Algorithm 7: An algorithm for solving MAXSAT using constrained hitting sets.

```
1 MAXHS-realizable (F)
2 Identical to Algorithm 5 except replace
3 hs = FindMinCostHittingSet(K)
4 by
5 hs = FindMinCostRealizableHittingSet(K)
```

there exists a truth assignment τ such that (a) for each clause $c \in hs$, $\tau \not\models c$, and (b) $\tau \models hard(\mathcal{F})$. Otherwise hs is said to be unrealizable.

An example of an unrealizable hitting set is one that contains clauses c_1, c_2 with a variable $x \in c_1$ and $\neg x \in c_2$, since all truth assignments satisfy either c_1 or c_2.

Any time the MCHS solver returns an unrealizable hitting set, at least one more iteration will be required before MAXHS terminates, as shown by the following corollary.

Corollary 1 (Of Theorem 3 on page 40)**.** *Let $\mathcal{K}$ be a set of cores of $\mathcal{F}$ and hs be a minimum cost hitting set of $\mathcal{K}$. If hs is unrealizable, then $\mathcal{F} \setminus hs$ is unsatisfiable.*

Proof. For contradiction, suppose $\pi \models \mathcal{F} \setminus hs$. Then $\pi \models hard(\mathcal{F})$ and since *hs* is unrealizable, π satisfies some clause in *hs*. So F_π the set of clauses falsified by π (a) is a strict subset of *hs* and (b) is a hitting set of $\mathcal{K}$. But then $cost(\mathcal{F}_\pi) < cost(hs)$ which contradicts the fact that *hs* is a minimum cost hitting set of $\mathcal{K}$. □

Therefore the MAXHS algorithm does not gain anything by encountering such unrealizable hitting sets. We can modify Algorithm 5 (page 41) to only use minimum cost realizable hitting sets, as shown in Algorithm 7. Theorem 3, upon which the correctness of MAXHS is based, still holds if minimum cost realizable hitting sets are computed rather than unconstrained minimum cost hitting sets.

Theorem 4. *If $\mathcal{K}$ is a set of cores for the* MAXSAT *problem $\mathcal{F}$, hs is a realizable hitting set of $\mathcal{K}$ that has minimum cost among all realizable hitting sets of $\mathcal{K}$, and π is a truth assignment satisfying $\mathcal{F} \setminus hs$ then $mincost(\mathcal{F}) = cost(\pi) = cost(hs)$.*

Proof. $mincost(\mathcal{F}) \leq cost(\pi) \leq cost(hs)$ by exactly the same argument as for Theorem 3 on page 40. Furthermore $mincost(\mathcal{F}) \geq cost(hs)$. Suppose for contradiction that $mincost(\mathcal{F}) < cost(hs)$ and let π be a solution to $\mathcal{F}$. Let $\mathcal{F}_\pi$ be the set of clauses falsified by π. Then $\mathcal{F}_\pi$ must be a hitting set of $\mathcal{K}$ and since there is a truth assignment π that falsifies every clause in $\mathcal{F}_\pi$ while also satisfying $hard(\mathcal{F})$, $\mathcal{F}_\pi$ is a realizable hitting set. But $cost(\mathcal{F}_\pi) = mincost(\mathcal{F}) < cost(hs)$. This is a contradiction since hs is the realizable hitting set of minimum cost. □

Observation 2. *Algorithm 7 correctly returns a solution to the inputted* MAXSAT *problem* $\mathcal{F}$.

Proof. The proof that Algorithm 5 is correct applies using Theorem 4 in place of Theorem 3. □

Realizability and Disjoint Cores Like in our basic MAXHS solver, we use a disjoint core phase before applying Algorithm 7. However, we do not enforce the realizability condition during the disjoint phase, because it is nontrivial to compute a minimal cost realizable hitting set even if the cores are disjoint. To see this, note that the realizability condition involves a SAT test over the hard clauses of the MAXSAT formula.

4.1.1 Implementing Realizability

In order to implement Algorithm 7, the realizability condition must be enforced by the hitting set solver. However, the realizability condition involves performing a SAT test to see if there is a truth assignment that falsifies all clauses in the hitting set and satisfies the hard clauses. We believe that a SAT solver is likely to outperform a MIP solver on this kind of problem, since it involves many logical constraints. We considered three possible approaches to implementing a search for realizable hitting sets. First, we could write our own realizable hitting set solver, based on Branch and Bound and incorporating SAT techniques. Second, we could augment the MIP model of the hitting set problem,

by introducing the original literals and clauses and relating them to the variables of the MIP model. Third, we could investigate using the callback interface of CPLEX so that the MIP model of the hitting set problem would remain unchanged, but as CPLEX progressed we could receive callbacks and verify the realizability condition using SAT techniques.

The second approach involves making the following changes to the CPLEX model to enforce the realizability condition. A new 0-1 variable is added to the MIP model for each of the original variables of $\mathcal{F}$ (assume they have the same names as in $\mathcal{F}$). Let x_i be the 0-1 variable of the MIP model that represents soft clause $C_i \in \mathcal{F}$, as defined in Section 3.5. We add to the MIP model the clauses $\{(\neg x_i \vee \neg \ell_{ij}) : \ell_{ij} \in C_i\}$, translated to linear constraints as specified before. Together, these constraints say that if x_i is set to one, then the clause C_i is falsified. Finally, we translate all hard clauses to linear constraints and add them to the MIP model. A solution to the resulting MIP will be a minimum cost realizable hitting set. However, we point out a potential disadvantage of this approach, which is that it involves adding all of the original hard clauses of the MAXSAT instance to the CPLEX model. We observed in Section 3.7.2 that CPLEX has difficulty solving many MAXSAT instances especially in the Industrial category, so we believe that adding these logical constraints to CPLEX will degrade the overall performance of MAXHS.

Of the three possible approaches to enforce the realizability condition, we begin by trying the first option, which is to implement a special-purpose Branch and Bound search. We do not investigate either of the other two proposed methods. However, as we will see in the following sections, in addition to realizability we pursue an alternative idea to constrain the hitting sets. This alternative approach leads to a very robust version of MAXHS. Therefore, we believe that returning to the question of how realizability should best be implemented is unlikely to be an interesting direction for future work.

We implemented a special-purpose Branch and Bound solver for the "FindMinCostRealizableHittingSet" function in Algorithm 7. The Branch and Bound solver searches for a minimum cost realizable hitting set by branching on whether or not a clause appears

in the hitting set. Each node corresponds to a partial hitting set. The cost of the best realizable hitting set found so far is maintained as the upper bound. At each node, some reasoning is performed to determine whether a better cost realizable hitting set can exist below the current node or if instead it is possible to backtrack. The realizability condition is checked whenever a leaf node is visited, and if the hitting set is realizable it becomes the new incumbent solution.

Dancing Links The Branch and Bound solver represents the current set of cores $\mathcal{K}$ using the dancing links data structure, which Knuth introduced to solve the Exact Cover problem[1] (Knuth, 2000). As the Branch and Bound search progresses (e.g. adding clauses to hitting set, hitting cores, and banning other clauses from the hitting set), the dancing links structure is easily updated and backtracked to represent the current subproblem. The main advantage of dancing links is that the effective size of the data structure changes in concert with the size of the current subproblem, so that near the leaves the hitting set representation is much smaller than it is at the root of the search tree. This means that any reasoning that needs to be performed on the current hitting set problem becomes cheaper as the search descends deeper, which can significantly reduce the total runtime since the deepest nodes are also the most numerous.

Subsumptions There are two well known simplification rules for the MCHS problem, that reduce the size of the problem while preserving at least one of the solutions (Weihe, 1998). The first rule removes a set κ if it supersets another set κ', since hitting κ' will also necessarily hit κ. The second rule says that an element c can be removed if there is another element c' such that $wt(c) \geq wt(c')$ and every set containing c also contains c'. The first rule is still sound for the problem of finding the minimum cost realizable hitting set, and so we apply this rule at every node of the Branch and Bound search in order to simplify the remaining problem. However, it is not sound to apply the second rule when enforcing the realizability condition, since it is no longer possible to know whether clause

[1]The Exact Cover problem is equivalent to the MCHS problem, except that each set can only be hit once.

c' is just as good as clause c, since realizability depends on the literals occurring in the clauses.

Lower Bound Branch and Bound search uses a lower bound calculation to perform lookahead and possibly prune the subtree below the current node. We use CPLEX to calculate the cost of a linear relaxation of the hitting set problem, ignoring the realizability condition. The linear relaxation is usually efficient to compute, and provides a good quality bound. This method provides a sound lower bound on cost of any minimum cost realizable hitting set extending the current partial hitting set. To see this, first recall that the cost of the linear relaxation of a MIP minimization problem is a lower bound on the cost of the optimum. We can ignore the realizability condition because the cost of a MCHS is always less than or equal to the minimum cost realizable hitting set.

SAT Checks As the Branch and Bound solver searches over partial hitting sets, it interacts with a SAT solver to check if the realizability condition is violated by the current hitting set. The SAT solver is initialized with just the hard clauses of the original MAXSAT formula, $\mathit{hard}(\mathcal{F})$, before Branch and Bound begins. When Branch and Bound adds a clause to the current hitting set, the negations of its literals are assigned in the SAT theory. Thus as Branch and Bound adds clauses to the partial hitting set, a partial assignment π is built up in the SAT solver. This is accomplished by enqueuing the literals at decision level zero in the SAT solver. Note that if the clauses in the hitting set can not all be falsified at the same time, then this will be caught immediately since it must be the case that some literal x has been assigned as well as its negation $\neg x$.

At each node of Branch and Bound, we use the following procedure to allow early detection that the hitting set is unrealizable. We ask the SAT solver to perform unit propagation, which will find some consequences of the partial assignment π that are implied by $\mathit{hard}(\mathcal{F})$. If unit propagation generates a conflict, it means that there is no realizable hitting set extending the current partial hitting set. Therefore, if unit propagation fails the Branch and Bound search can backtrack.

If the Branch and Bound search reaches a leaf node, all cores have been hit. We know that the clauses in the hitting set can all be falsified because of the unit propagation checks that have been performed at each node along the current path. However, we are not sure that the current assignment π can be extended to a solution of $hard(\mathcal{F})$, since only unit propagation has been applied and not a full SAT check. So at each leaf node, we ask the SAT solver to solve the hard clauses, by extending the partial assignment π. If the SAT solver finds a solution, it shows that the hitting set is realizable, and otherwise the hitting set is unrealizable.

Future Work There are still a number of possible improvements to the Branch and Bound search that could be implemented, such as OR-Decomposition (Kitching and Bacchus, 2008), caching, and alternate lower bounding techniques like Lagrangian relaxation (Wolsey, 1998).

4.1.2 Experiments with Realizability

Our Branch and Bound hitting set solver is much slower than CPLEX. However, on some problems the ability to enforce the realizability condition makes up for the inefficiency of our implementation. We report interesting results obtained from some experiments on instances from the 2009 MAXSAT Evaluation. We found four benchmark families for which enforcing the realizability condition paid off. In particular, Algorithm 7 solves 44 instances that Algorithm 5 can not solve. These instances are shown in Table 4.1, which lists the number of instances solved by Algorithm 7 (column '#'), the average cost of their optima (column 'Avg OPT'), the average number of iterations Algorithm 5 performed before the timeout of 1200 seconds, and the number of iterations and runtime for Algorithm 7. Observe that the number of iterations that Algorithm 7 requires to solve the problem is usually significantly fewer than Algorithm 5 performs, even though Algorithm 5 fails to solve these instances. This demonstrates that constraining the hitting sets to be realizable can sometimes reduce the number of iterations enough to significantly

		Avg	Alg. 5	Alg. 7	
Family	#	OPT	Iter	Iter	Time
ms/Sean	4	1	13	67	434
pms/bcp-msp	26	99	460	121	204
pms/bcp-mtg	13	8	2198	757	258
pms/bcp-syn	1	6	80	53	295

Table 4.1: Results on instances Algorithm 7 can solve within 1200 s but Algorithm 5 cannot.

improve the total runtime required to solve the problem.

In Table 4.2, more detailed results are presented on 13 instances from the Industrial partial MAXSAT family bcp-syn. These MAXSAT instances share a common structure: all of their hard clauses contain only positive literals, and all of their soft clauses are unit clauses containing negative literals. Thus, the original MAXSAT instances themselves represent MCHS problems. These problems have reasonably large optima, requiring between 17 and 287 clauses to be relaxed. None of the existing MAXSAT solvers we experimented with (i.e., the set used in Section 3.7) could solve these problems within the timeout, and furthermore these instances were not solved by any solver in the 2009 and 2010 MAXSAT Evaluations. However, CPLEX is able to solve each of these problems quite quickly, as shown in column "CPLEX Time". The table also shows the results of using MAXHS with and without the realizability condition. Algorithm 7 as described in Section 4.1.1 is shown in columns "Alg. 7 - B&B", and Algorithm 5 with CPLEX for the hitting set solver is shown in columns "Alg. 5 - CPLEX". We also experimented with a version of our Branch and Bound solver that does not enforce the realizability condition, and the results are shown in columns "Alg. 5 - B&B". For each version of the algorithms, and each instance, the number of iterations, the average time to solve the hitting set problems (columns 'HS'), and the total runtimes are given in the table. The table also shows some information about the size of the hitting set problems encountered. Column '|Core|' reports the average number of clauses in the cores. Column 'MxN' reports the average dimensions of the hitting set problem given to CPLEX after the subsumption

rules (defined in Section 4.1.1) have been applied. The value 'M' is the average number of cores in the hitting set problems, and 'N' is the average size of the cores in the hitting set problems. The 'Nodes' columns give the average number of nodes searched by B&B while solving the hitting set problems. The time the SAT solver takes to generate each core is always less than 0.02 seconds, so this time is not included in the table.

We can observe that the MAXHS approach is better suited to the structure of the MCHS problem than any existing MAXSAT solvers. The cores of a MCHS problem correspond to the sets to hit. That is, every set to hit κ is specified by a hard clause in the MAXSAT instance. So if the MAXSAT instance contains a hard clause (x, y, z) representing set κ, then the soft clauses $\{(\neg x), (\neg y), (\neg z)\}$ are a minimal core corresponding to κ. The SAT solver will discover these cores one at a time and pass them to CPLEX. Eventually, the CPLEX model will grow to encompass the entire original MCHS problem, at which point the hitting set returned by CPLEX will correspond to the MAXSAT solution. The number of cores that must be passed to CPLEX should be no more than the number of sets to hit in the original MCHS problem, i.e. the number of hard clauses. So the number of iterations of MAXHS should only be linear, and each iteration will be efficient if CPLEX performs well on the original hitting set problem. We can see from Table 4.2 that the number of cores required by MAXHS is always less than the number of hard clauses in each instance (column "# Hard"), as expected.[2]

However, the realizability condition does not have any affect on these MAXSAT problems that encode MCHS. This is because every hitting set is realizable: falsifying the clauses in the hitting set only sets variables to *true*, and such an assignment can always be extended to a solution to the hard clauses since the hard clauses only contain positive literals. This explains why Algorithm 7 does the worst of the three MAXHS algorithms on the set of instances in Table 4.2; there is no compensation for the overhead of checking

[2]Note that in an instance of the MCHS problem, it may be possible to remove some of the sets to hit without changing the solution to the MCHS instance. This explains why the number of cores required by MAXHS can be significantly less than the number of hard clauses in the MAXSAT instance.

			CPLEX	Alg. 5 - CPLEX					Alg. 5 - B&B				Alg. 7 - B&B			
Instance	# Hard	OPT	Time	Iter	|Core|	MxN	HS	Time	Iter	Nodes	HS	Time	Iter	Nodes	HS	Time
saucier.r	116	6	219	80	2885	40x2167	15	-	3		1195	-	53	5	5	295
300_10_20	100	17	4	96	14	48x147	1	148	94	14	1	142	97	17	1	152
300_10_14	100	19	0	93	11	46x130	0	35	89	12	0	46	88	16	0	37
300_10_15	100	19	4	99	12	49x144	0	62	95	13	1	106	96	17	1	134
300_10_10	100	21	0	95	9	47x119	0	13	95	14	0	47	95	18	0	42
ex5.r	690	37	3	285	28	132x294	0	116	281	24	4	1187	260	46	5	-
ex5.pi	718	65	3	304	25	137x267	0	72	301	23	3	924	271	50	2	511
test1.r	305	110	0	278	6	119x176	0	3	277	9	0	17	271	85	0	67
bench1.pi	364	121	0	330	8	149x290	0	25	331	28	1	298	328	113	1	264
max1024.r	964	245	2	747	5	323x377	0	153	734	28	1	1016	635	179	2	-
max1024.pi	978	259	3	724	5	310x358	0	200	720	25	2	-	663	187	2	-
prom2.r	1610	278	0	935	6	385x498	0	61	968	21	0	717	733	225	2	-
prom2.pi	1619	287	0	914	6	372x484	0	40	966	26	1	846	747	249	2	-

Table 4.2: Detailed results on 13 instances from the industrial PMS bcp-syn family. '-' in the Time columns indicates timeout after 1200 seconds.

the realizability condition.

4.2 Non-Core Constraints

In this section we propose an alternative paradigm for constraining the hitting sets used by MAXHS. In the previous section we observed that the logical structure of the soft and hard clauses of the original MAXSAT formula impose many constraints on how the soft clauses can be falsified. Section 4.1 introduced the realizability condition to express such constraints. However, we will show that the realizability condition does not capture all of the possible constraints on the hitting sets. In this section we show how more of these constraints can be learned by the SAT solver, and enforced by the MIP solver. In contrast to the realizability condition, the constraints we introduce in this section are efficiently handled by CPLEX. Unlike the constraints discovered by the SAT solver in Chapter 3, these constraints do not correspond to cores; hence we will call the additional constraints non-core constraints. It can be noted that core constraints always contain only positive b-variables, indicating that at least one of the corresponding soft clauses

must be falsified. In contrast, non-core constraints will be clauses over the b-variables that contain at least one negative b-variable.

Recall that MAXHS uses b-variables with the assumption mechanism of the SAT solver to generate cores, and the b-variables are also used by the MIP model to represent the hitting set problem. The technique we propose relies on strengthening the relationship between the b-variables and the original clauses they relax, to capture the full meaning of the b-variables: setting one to *true* is equivalent to falsifying its corresponding clause. By augmenting the b-variable relaxation with these equivalences, the SAT solver can be utilized to learn more general conditions on how the soft clauses can be falsified, conditions that are not expressible using core constraints or realizability alone.

4.2.1 b-variable Equivalences

Many sound constraints exist over the b-variables that do not take the form of core constraints, as illustrated by the following example.

Example 5. *Let* $\mathcal{F} = \{(x),\ (\neg x),\ (x, y),\ (\neg y),\ (\neg x, z),\ (\neg z, y)\}$ *where each clause has weight 1.* $\mathcal{F}^b$ *is therefore the set of clauses* $\{(b_1, x),\ (b_2, \neg x),\ (b_3, x, y),\ (b_4, \neg y),\ (b_5, \neg x, z),\ (b_6, \neg z, y)\}$. *Suppose that the three cores* $\kappa_1 = \{(x),\ (\neg x)\}$, $\kappa_2 = \{(\neg x),\ (x, y),\ (\neg y)\}$, *and* $\kappa_3 = \{(x, y),\ (\neg y),\ (\neg x, z),\ (\neg z, y)\}$ *have been found. These cores correspond to the core constraints* $\mathcal{K} = \{(b_1, b_2),\ (b_2, b_3, b_4),\ (b_3, b_4, b_5, b_6)\}$. *We see that to satisfy these core constraints at least two b-variables must be set to true, and at least two soft clauses will be falsified by the* MAXSAT *solution. When* CPLEX *searches for a MCHS of* $\mathcal{K}$, *as soon as* b_1 *is assigned,* $\neg b_2$ *could be inferred because it is impossible to falsify both* (x) *and* $(\neg x)$ *at the same time. Similarly, whenever* $\neg b_2$ *is assigned we could obtain* $\neg x$ *and* b_1 *by unit propagation in* $\mathcal{F}^b \cup \mathcal{K}$. *However, we do not detect that* $\neg b_5$ *must hold as well since its soft clause is now satisfied. These two examples demonstrate that in addition to the core constraints* $\mathcal{K}$, $\mathcal{F}^b$ *also implies the constraints* $(\neg b_1, \neg b_2)$ *and* $(b_2, \neg b_5)$. *If these constraints could be discovered automatically, then the search for a MCHS could be*

further constrained and potentially made more efficient.

This example shows that the realizability condition does not capture all sound constraints over the b-variables. Although the realizability condition would enforce the first non-core constraint in Example 5, $(\neg b_1, \neg b_2)$, it would not capture the second, $(b_2, \neg b_5)$. Therefore, we must look beyond the realizability condition for techniques to discover non-core constraints that the b-variables must satisfy.

Example 5 indicates that although the b-variables of $\mathcal{F}^b$ are intended to represent the soft clauses of $\mathcal{F}$ this correspondence is not strictly enforced by $\mathcal{F}^b$. That is, $\mathcal{F}^b$ admits models (satisfying truth assignments) that unnecessarily set b-variables to *true* even when the corresponding soft clause is satisfied. Proposition 7 shows, however, that minimum cost models of $\mathcal{F}^b$ do obey a stricter correspondence of equivalence between the b-variable settings and the soft clauses satisfied. Since MAXSAT solving involves searching for minimum cost models, a natural and simple modification to $\mathcal{F}^b$ is to force the b-variables to be equivalent to the negation of their corresponding soft clauses.

Definition 17 ($\mathcal{F}^b_{eq}$). *Let $\mathcal{F}$ be a* MAXSAT *formula. Then*

$$\mathcal{F}^b_{eq} = \mathcal{F}^b \cup \bigcup_{c_i \in soft(\mathcal{F})} \{(\neg b_i, \neg \ell) : \ell \in c_i\}$$

is the relaxed SAT instance of $\mathcal{F}$ with b-variable equivalences.

It is possible to define a correspondence between the truth assignments for $\mathcal{F}$ and the truth assignments for $\mathcal{F}^b_{eq}$.

Definition 18 (π^b). *If π is a truth assignment to the variables of $\mathcal{F}$, π^b is the truth assignment to the variables of $\mathcal{F}^b_{eq}$, where*

$$\pi^b = \pi \cup \{\neg b_i : \pi \models c_i, c_i \in soft(\mathcal{F})\} \cup \{b_i : \pi \not\models c_i, c_i \in soft(\mathcal{F})\}.$$

If π^b is a truth assignment to the variables of $\mathcal{F}^b_{eq}$, then π denotes the truth assignment

to the variables of $\mathcal{F}$ that is π^b restricted to the variables of $\mathcal{F}$.

In this definition π^b is constructed so that it assigns each b-variable a truth value equivalent to the negation of the truth value π assigns to the corresponding soft clause. Thus π^b models the b-variable equivalences. Under this mapping there is a 1-1 correspondence between the models of $\mathcal{F}^b_{eq}$ and the models of $hard(\mathcal{F})$.

Proposition 9. *Let π be a truth assignment to the variables of $\mathcal{F}$. Then $\pi \models hard(\mathcal{F})$ if and only if $\pi^b \models \mathcal{F}^b_{eq}$.*

Proof. Suppose $\pi \models hard(\mathcal{F})$. Consider a clause in $\mathcal{F}^b_{eq}$, we show that it is satisfied by π^b. There are two cases depending on whether the clause also appears in $\mathcal{F}^b$ or if it is one of b-variable equivalence clauses. If the clause also appears in $\mathcal{F}^b$, and it does not contain a b-variable, then it is an original hard clause and is therefore satisfied by π and by π^b. Otherwise, the clause is of the form $C_i \cup \{b_i\}$ and either it is satisfied by π^b because $\pi \models C_i$, or if $\pi \not\models C_i$, then it is satisfied by π^b because by the definition of π^b if $\pi \not\models C_i$ then π^b assigns b_i to *true*. In the second case, the clause in $\mathcal{F}^b_{eq}$ is of the form $\{\neg b_i, \neg \ell\}$ for some original literal $\ell \in C_i$. Suppose that $\{\neg b_i, \neg \ell\}$ is falsified by π^b. Then π^b assigns b_i to *true*, and by the definition of π^b it must be the case that $\pi \not\models C_i$. But then $\pi \models \neg \ell$ which satisfies the clause and thus we have reached a contradiction. So in every case, we have shown that π^b satisfies the clause of $\mathcal{F}^b_{eq}$. Therefore $\pi^b \models \mathcal{F}^b_{eq}$.

On the other hand, if $\pi^b \models \mathcal{F}^b_{eq}$ then we immediately have that $\pi \models hard(\mathcal{F})$ since $hard(\mathcal{F}) \subseteq \mathcal{F}^b_{eq}$. So we have proven that $\pi \models hard(\mathcal{F})$ if and only if $\pi^b \models \mathcal{F}^b_{eq}$. □

Proposition 10. *Let π and π^b be two truth assignments defined according to Definition 18. If $\pi^b \models \mathcal{F}^b_{eq}$ then $cost(\pi) = bcost(\pi^b)$.*

Proof. The sum of the weights of the clauses corresponding to the b-variables set to *true* in π^b make up $bcost(\pi^b)$. Since $\pi^b \models \mathcal{F}^b_{eq}$, by Proposition 9 we know that $\pi \models hard(\mathcal{F})$ and therefore $cost(\pi)$ is equal to the sum of the weights of the soft clauses of $\mathcal{F}$ falsified

by π. By Definition 18, the b-variables set to *true* by π^b are exactly those that correspond to the soft clauses of $\mathcal{F}$ falsified by π. Therefore, $cost(\pi) = bcost(\pi^b)$. □

Proposition 11. *π is a solution for the* MAXSAT *formula $\mathcal{F}$ if and only if π^b achieves minimum bcost over all satisfying truth assignments for $\mathcal{F}^b_{eq}$.*

Proof. This follows immediately from Proposition 10. Every satisfying assignment for $\mathcal{F}^b_{eq}$ gives an assignment for $\mathcal{F}$ that has equal cost. Thus the MAXSAT solution has cost at most the cost of the minimal *bcost* assignment for $\mathcal{F}^b_{eq}$. On the other hand, the MAXSAT solution can not have strictly smaller cost than this, since if it did then the MAXSAT solution could be extended to a satisfying assignment of $\mathcal{F}^b_{eq}$ with equal *bcost*. □

Proposition 11 shows that the MAXSAT instance $\mathcal{F}$ can be solved by searching for a *bcost* minimal satisfying assignment to $\mathcal{F}^b_{eq}$.

4.2.2 MAXHS with Non-Core Constraints

The extension of Algorithm 5 (page 41) to utilize non-core constraints is conceptually simple. The encoding $\mathcal{F}^b_{eq}$ is simply substituted for the weaker encoding $\mathcal{F}^b$. Now since in $\mathcal{F}^b_{eq}$ the b-variables are no longer pure, the SAT solver can return both core and non-core constraints. Each constraint is passed to the MIP solver which operates as before. (A copy of the learnt constraint is also kept by the SAT solver because it should help the SAT solver to prune the search space in future invocations). The resulting modified version of Algorithm 5 is shown in Algorithm 8.

Initially, the set of b-variable constraints (clauses), $\mathcal{K}$, is empty (line 2). The objective function is defined on line 3 as the sum of the clause weights for b-variables that are assigned *true*. On line 5, an assignment to the b-variables, $\mathcal{A}$, is calculated that satisfies the current constraints $\mathcal{K}$ and minimizes the value of the objective function *obj*. This setting of the b-variables is passed as the set of assumptions to the SAT solver on line 6, along with the SAT instance $\mathcal{F}^b_{eq}$. If the SAT solver returns UNSAT, κ will be a clause

Algorithm 8: A MAXSAT algorithm that exploits non-core constraints.

```
1  MAXSAT-solver (F)
2  K = ∅
3  obj = wt(c_i) * b_i + ... + wt(c_k) * b_k
4  while true do
5      A = Optimize(K,obj)
6      (sat?,κ) = AssumptionSatSolver(F^b_eq, A)
       ; // If SAT, κ contains the satisfying truth assignment.
       ; // If UNSAT, κ contains a clause over b-variables.
7      if sat? then
8          break ; // Finished, κ is a MAXSAT solution.
       // Add new constraint to the optimization problem
9      K = K ∪ {κ}
       // And to the SAT formula for better performance
10     F^b_eq = F^b_eq ∪ {κ}
11 return (κ, bcost(κ))
```

over negated literals from $\mathcal{A}$. This constraint is added to $\mathcal{K}$ on line 9 and the process iterates until the SAT solver reports a solution.

Theorem 5. *Algorithm 8 returns a solution to the inputted* MAXSAT *problem* $\mathcal{F}$.

Proof. First, observe that if the κ returned by the SAT solver at line 6 is a clause then $\mathcal{F}^b_{eq} \models \kappa$. Furthermore, if κ is a satisfying assignment then $bcost(\kappa)$ is equal to the sum of the costs of the true b-variables in $\mathcal{A}$, the solution returned by the optimizer at line 5. This follows from the fact that κ extends $\mathcal{A}$ which has already set all of the b-variables. Let κ be the satisfying truth assignment causing the algorithm to terminate. All satisfying assignment of $\mathcal{F}^b_{eq}$ satisfy the constraints in $\mathcal{K}$ as each of these is entailed by $\mathcal{F}^b_{eq}$. Furthermore, $bcost(\kappa)$ is equal to the cost of an optimal solution to these constraints, thus κ achieves minimal *bcost* over all satisfying truth assignments for $\mathcal{F}^b_{eq}$, and by Proposition 11 κ restricted to the variables of $\mathcal{F}$ is a MAXSAT solution for $\mathcal{F}$.

Second, observe that each iteration except the final one adds a constraint to $\mathcal{K}$ that eliminates at least one more setting of the b-variables. Since there are only a finite number of different settings, the algorithm must eventually terminate. □

The key difference with Algorithm 5 is that the optimizer no longer deals with pure hitting set problems as the constraints can now contain negative b-variables. This means that non-core constraints change the paradigm of MAXHS from an implicit hitting set problem. It now becomes more like a logic based Benders decomposition approach (Hooker, 2007). In particular, the optimization problem is being solved only over the b-variables while the SAT solver is being used to add additional constraints to the optimization model until the solution also satisfies the feasibility conditions. Although CPLEX is no longer solving a hitting set problem, it remains very effective in the presence of non-core constraints.

4.2.3 Seeding CPLEX with Constraints

Each call to CPLEX's solve routine incurs some overhead so it is desirable to reduce the number of calls to CPLEX. One way to accomplish this is by "seeding" the MIP model with a number of initially computed b-variable constraints. In this way each candidate solution (setting of the b-variables) returned by CPLEX is more informed about the constraints of the problem and thus more likely to be a true solution. We perform seeding after the disjoint core phase, but before the iterations of Algorithm 8 begin. Below, we describe several techniques to cheaply identify such additional b-variable constraints.

Equivalence Seeding: In $\mathcal{F}^b_{eq}$, literals that appear in soft *unit* clauses of $\mathcal{F}$ are actually logically equivalent to their b-variables. To see this, recall that if $c_i = (x) \in \mathit{soft}(\mathcal{F})$ is a soft unit clause, then $\mathcal{F}^b_{eq}$ will contain clauses (x, b_i) and $(\neg b_i, \neg x)$. These two clauses together imply that $b_i \equiv \neg x$. For a clause c of $\mathcal{F}^b$, if each variable in c has an equivalent b-variable (or is itself a b-variable), then we can derive a new constraint from c by replacing every original variable by its equivalent b-variable. This constraint is a clause over the b-variables that can be added to CPLEX.

Example 6. *In Example 5, $b_1 \equiv \neg x$ due to the soft unit clause (x) and its relaxation by b_1. Similarly, $b_4 \equiv y$. Therefore, from the relaxed clause $(b_3, x, y) \in \mathcal{F}^b$ we can obtain the*

b-variable constraint $(b_3, \neg b_1, b_4)$ *by simply substituting the equivalent b-variable literals for the original literals.*

Implication Seeding: In $\mathcal{F}^b_{eq}$, each of the b-literals may imply other b-literals. We perform a trial unit propagation on each b-literal b_i in order to collect a set of implied b-literals $imp(b_i) = \{b_i^1, ..., b_i^k\}$. This represents a conjunction of k binary clauses $b_i \rightarrow b_i^j$ $(1 \leq j \leq k)$ over the b-variables. Although these k clauses could be individually added to CPLEX we can in fact encode their conjunction in a single linear constraint that can be given to CPLEX:

$$-k \times b_i + b_i^1 + \cdots + b_i^k \geq 0$$

Note that these are b-literals, so as is standard a negative literal b is encoded as $(1 - var(b))$ and a positive literal is encoded as $var(b)$ (Section 3.7). To understand this constraint note that if b_i is *true* (equal to 1) then all of the b_i^j literals must be 1 to make the sum non-negative.

Implication+Reverse Seeding: During the trial unit propagation of each b-literal b_i, we can also keep track of every original literal that is found to be implied by b_i in order to obtain sets of reverse implications: $rev(x) \subseteq \{b_i : \mathcal{F}^b_{eq} \wedge b_i \models x\}$. Then, for each clause $c_i \in \mathcal{F}^b$, we check if each of its original literals $x \in c_i$ has a non-empty $rev(\neg x)$. If so, a b-literal $b_{\neg x} \in rev(\neg x)$ is chosen for each x and its negation $\neg b_{\neg x}$ is substituted for x in c_i. The result is a new clause containing only b-variables that can be added to CPLEX. It is easy to see that this clause is sound by considering the following example.

Example 7. *Suppose that* $(x, y, b_1) \in \mathcal{F}^b_{eq}$ *where* x, y *are original variables and* b_1 *is a relaxation variable. Suppose that* $b_{\neg x} \in rev(\neg x)$ *and* $b_{\neg y} \in rev(\neg y)$*. This means that clauses* $(\neg b_{\neg x}, \neg x)$ *and* $(\neg b_{\neg y}, \neg y)$ *are implied by* $\mathcal{F}^b_{eq}$*. Therefore* $(\neg b_{\neg x}, \neg b_{\neg y}, b_1)$*, which can be obtained in two resolution steps, is also implied by* $\mathcal{F}^b_{eq}$ *and can be added to* CPLEX*.*

Since the b-literal implications $imp(b_i)$ are also available, we can add the Imp-Seeding

constraints as well if we are computing the Rev-Seeding constraints. Note that if $b \equiv x$, as in Eq-Seeding, we obtain at least as many seeded constraints as would be obtained by Eq-Seeding. If $rev(\neg x)$ contains more than one b-literal, we could choose any one of them to form the new clause. We simply use an equivalent b-literal if one exists, and otherwise we choose the first b-literal that was found to imply $\neg x$. In future work we could investigate different ways of choosing the member of $rev(\neg x)$, or methods for using them all.

4.2.4 Implementation

The implementation of Algorithm 8 is based on the implementation of Algorithm 5. Here, in the initial disjoint core phase (see Section 3.5), the SAT solver considers only $\mathcal{F}^b$ and can ignore the b-variable equivalences. After the disjoint core phase, the b-variable equivalences are added to the SAT instance to obtain $\mathcal{F}^b_{eq}$. After the b-variable equivalences are added to the SAT formula, CPLEX is seeded with any b-variable constraints obtained from the input clauses (Section 4.2.3).

4.2.5 Experimental Results

In this section we examine the empirical behaviour of Algorithm 8 and the various kinds of seeding described in Section 4.2.3. The experimental setup is the same as in Section 3.7. All of the techniques introduced in this chapter were implemented on top of the best version of MAXHS from the previous chapter, MAXHS-min-disj-inv-del. Therefore, we will omit the designations "min-disj-inv-del" when naming the versions of MAXHS that include techniques from the current chapter. We report results with five versions of MAXHS that use Algorithm 8 and different types of seeding described in Section 4.2.3. The five versions are listed in Table 4.3.

Overall Performance

We compare the overall performance of the different solvers in terms of the number of

MAXHS Version	Min Cores	Disjoint Phase	Invert Activity	Delete Learnts	Non-Cores (Alg. 8)	Equiv Seeding	Implication Seeding	Reverse Seeding
MAXHS-noncore	✓	✓	✓	✓	✓			
MAXHS-noncore-eq	✓	✓	✓	✓	✓	✓		
MAXHS-noncore-imp	✓	✓	✓	✓	✓		✓	
MAXHS-noncore-rev	✓	✓	✓	✓	✓			✓
MAXHS-noncore-imp-rev	✓	✓	✓	✓	✓	✓	✓	✓

Table 4.3: The five versions of MAXHS that we evaluate in this chapter.

problems they solve within the time limit. The overall results are shown in two cactus plots. The first plot, Figure 4.1, compares the competing solvers and the best version of MAXHS from the previous chapter with the best version of MAXHS from this chapter, MAXHS-noncore-eq. The second plot, Figure 4.2 shows the results for each of the five new versions of MAXHS, and the best version of MAXHS from the previous chapter for reference.

We observe that seeding CPLEX with constraints is very beneficial to the performance of MAXHS. Indeed, the best version, MAXHS-noncore-eq, outperforms CPLEX and all other MAXSAT solvers, as shown in Figure 4.1. The techniques introduced in this chapter move the overall performance of the MAXHS approach from behind the top three MAXSAT solvers to well in front of them. These results confirm that giving more information to CPLEX allows MAXHS to benefit from the excellent performance of CPLEX on some MAXSAT instances that was observed in Section 3.7.2.

The results in Figure 4.2 show the overall performance of the various seeding policies. It appears that Algorithm 8 is actually slightly less robust than Algorithm 5 from Chapter 3. The boost in performance only occurs when some type of seeding is also employed. Among the four different types of seeding, we see that in general the policies that identify more constraints to seed CPLEX with work better. That is, MAXHS-noncore-imp-rev solves more problems than MAXHS-noncore-rev or MAXHS-noncore-imp. However, MAXHS-noncore-eq is even a bit better than MAXHS-noncore-imp-rev, even though the seeding performed by MAXHS-noncore-imp-rev should subsume that done by MAXHS-noncore-eq as explained in Section 4.2.3.

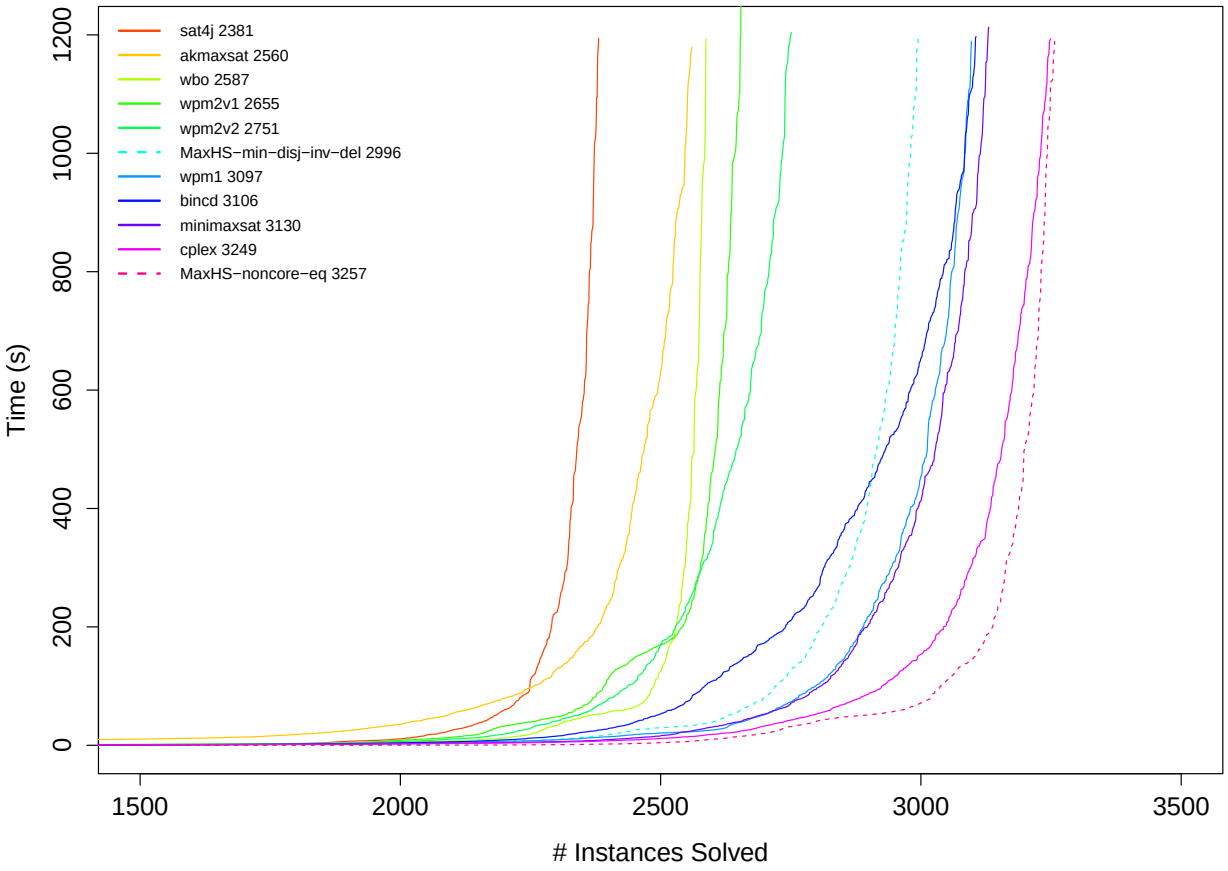


Figure 4.1: Runtime results for the competing solvers, the overall best version of MAXHS from Chapter 3 (MAXHS-min-disj-inv-del) and the overall best version of MAXHS with non-cores and seeding (MAXHS-noncore-eq). Shows how many problems were solved within each time limit. The total number of instances solved is given in the legend after the solver's name.

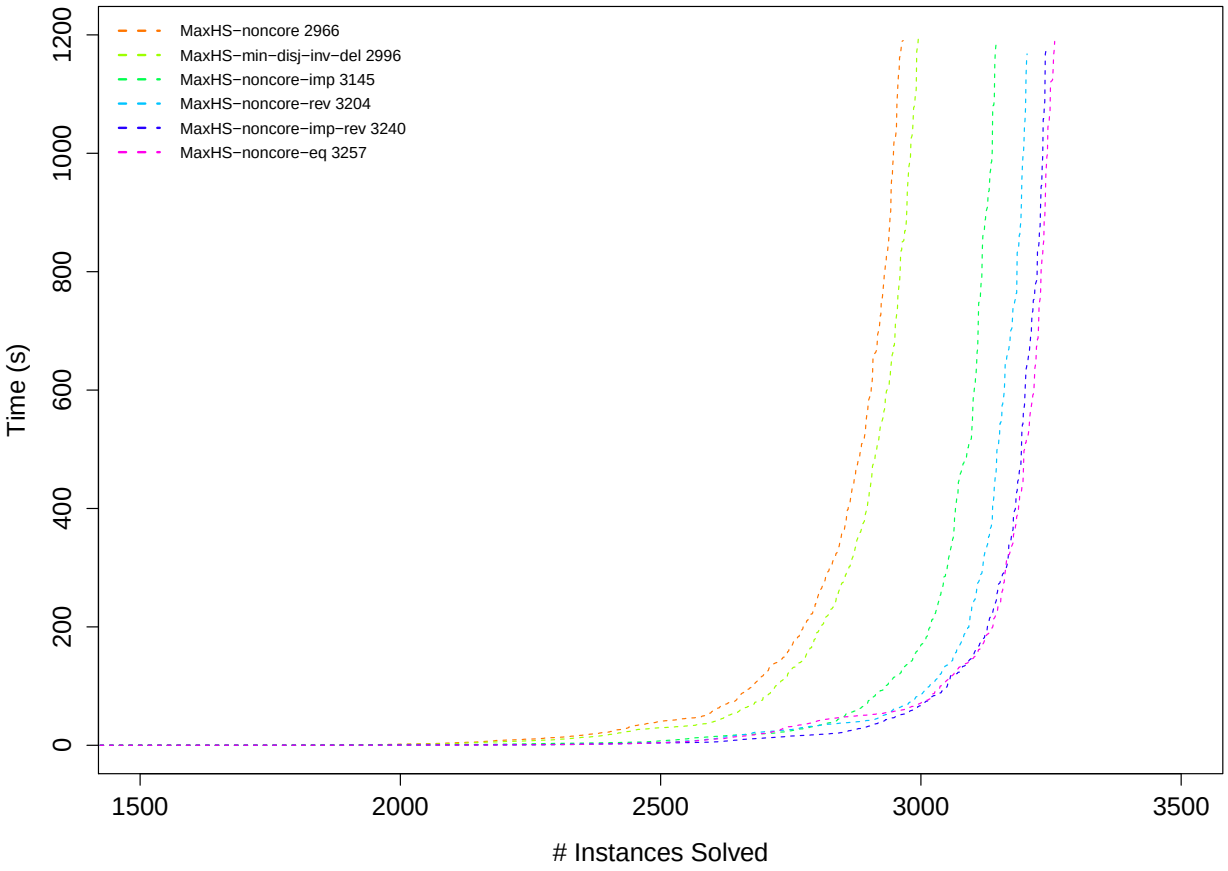


Figure 4.2: Runtime results for the overall best version of MAXHS from Chapter 3 (MAXHS-min-disj-inv-del) and five versions of MAXHS with non-cores and seeding. Shows how many problems were solved within each time limit. The total number of instances solved is given in the legend after the solver's name.

Tables 4.4 and 4.5 show the number of problems solved broken down by benchmark family and Crafted vs. Industrial categories. We observe that seeding helps on both Crafted and Industrial instances, and that the MAXHS approach is usually able to solve a good number of instances in every benchmark family. Thus the MAXHS approach with non-core constraints and seeding appears to work well on many different types of problems. These results also show that the best method of seeding varies considerably for different problems. This suggests that it is possible to develop even better techniques to determine which constraints to seed CPLEX with, and how many.

Trade-offs in MAXHS

Next, we focus on the behaviour of MAXHS when the techniques of non-core constraints and seeding are employed. As described in Section 3.7.2, we collected statistics from the runs of each version of our solver on each instance, including the number of calls to CPLEX's solve routine, the total time spent by CPLEX in calls to its solve routine, the total time spent in MINISAT, and the cost of the last lower bound. In Figures 4.3-4.8, we compare versions of MAXHS along four dimensions: the runtime on each instance, the CPLEX time per CPLEX call, the SAT time per CPLEX call, and the increment in the cost of the lower bound per CPLEX call. All 4502 instances appear in each of these graphs, although the number of data points sometimes appears to be fewer because they overlap.

The main observation from these graphs is that the effect of seeding is to significantly increase the increment in the lower bound per call to CPLEX. We see in Figures 4.4- 4.6(b) that the lower bound increment per CPLEX call is always above the 45° line. This means that the constraints we seed CPLEX with are actually very informative, as we hoped. The seeded constraints help to immediately rule out many lower cost assignments that CPLEX would otherwise return. However, the size of the CPLEX models is also significantly larger due to the initial seeding, and as might be expected, this does increase the time to solve each CPLEX problem (see Figures 4.4-4.8(d)). Yet the amount of time spent in SAT solving is not significantly affected either by using non-core constraints, or by seeding (Figures 4.3-4.8(c)).

Overall, these results show that our method of seeding CPLEX with constraints allows very informative constraints to be added to the CPLEX problem, without requiring additional calls to the SAT solver. Even though the CPLEX problems get harder to solve, fewer calls to CPLEX are necessary because the seeded constraints rule out many low-cost assignments immediately. These results also show that seeding tends to make the CPLEX

models harder to solve, and sometimes a previously solvable instance becomes impossible to solve within the time limit (although the opposite occurs more frequently). This explains in part why the weakest type of seeding (Equivalence seeding, implemented in maxhs-noncore-eq) ends up solving slightly more problems overall than the most extensive form of seeding (Implication+Reverse seeding, implemented in maxhs-noncore-imprev). Clearly, in future work better methods for deciding which level of seeding to apply on an instance specific basis could result in considerable performance gains.

4.3 Conclusions

In this chapter we were motivated by observing that the maxsat problem imposes many additional constraints on the solutions to the hitting set problems. These constraints no longer take the form of cores, so we showed how maxhs can learn such non-core constraints via a natural modification to the SAT formulas. By adding more of these constraints to the cplex model, many low-cost assignments that cplex would otherwise return are eliminated.

Furthermore, we showed how to use the logical relationship between the relaxation variables and the original clauses in order to quickly identify many non-core constraints that we can use to seed the cplex model. This technique leads to dramatic performance improvements. The resulting version of our solver, maxhs-noncore-eq, solves more problems than any other maxsat solver including cplex itself. The robustness of the maxhs approach is also significantly improved.

The ideas proposed in this chapter raise one main open question. The use of b-variable equivalences and more general clausal constraints over them sets the stage for an investigation into how the two solvers (MIP and SAT) should best be combined. For example, in the current algorithm, the two solvers communicate over a restricted shared language consisting of only the b-variables, but we could consider expanding this language

to include some or all of the original variables. Another example is that in the current algorithms, the CPLEX model contains mostly clausal constraints (except if seeding is used). Investigating ways to further exploit the ability of the MIP solver to handle general linear constraints could lead to a more powerful hybrid optimization solver.

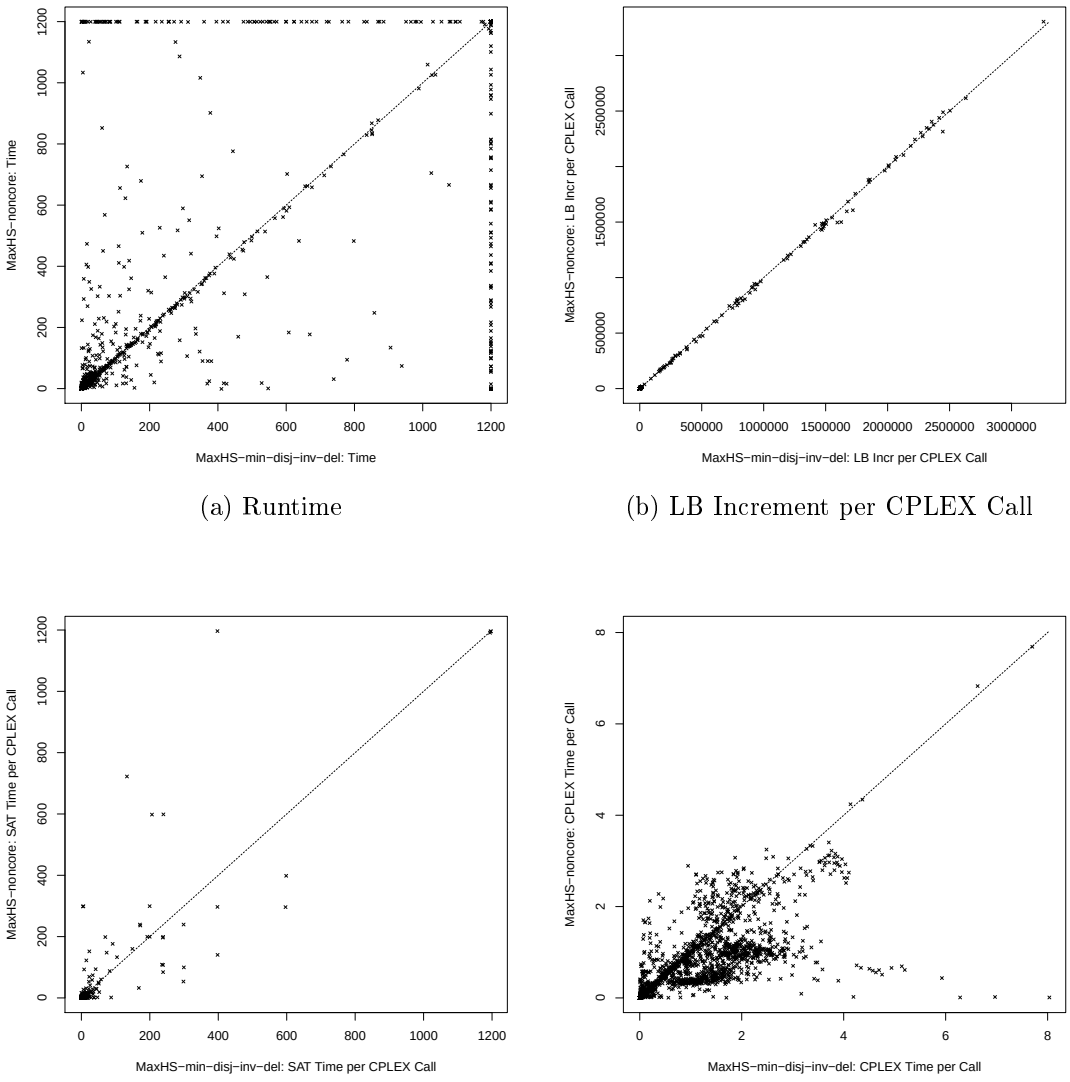
In addition, non-core constraints with b-variable equivalences bring the MAXHS solver much closer to a Benders Decomposition approach. This gives rise to a number of questions about exactly where the division of labour should lie between the optimization solver and the feasibility solver. Future work may investigate other ways of dividing the work between these two solvers with the aim of improving overall performance.

Family	#	mini	MaxHS ch3	MaxHS noncore	MaxHS noncore imp	MaxHS noncore eq	MaxHS noncore rev	MaxHS noncore imp rev
ms/spinglass/	20	20	0	0	0	0	0	0
wms/kexu/frb-wcnf/	35	15	10	10	10	20	20	18
pms/csp/sparseloose	20	20	20	11	10	11	10	10
wpms/pb/factor/	186	186	186	186	186	186	186	186
pms/csp/denseloose	20	20	10	0	0	0	0	0
KnotPipatsrisawat	350	117	61	53	69	52	41	64
pms/queens	7	7	5	3	3	3	3	3
wpms/aucregions	84	84	35	35	84	84	84	84
ms/cut/spinglass	5	3	1	1	1	1	1	1
pms/jobshop	4	2	4	4	4	3	4	4
wpms/planning	71	71	69	71	71	71	71	71
pms/maxone/struc	60	60	46	55	55	60	58	60
ms/ramsey	48	35	34	34	33	34	34	33
pms/clique/rand	96	96	4	4	96	96	96	96
wms/cut/spinglass	5	4	1	1	1	1	1	1
wpms/pb/miplib	16	5	7	7	7	7	7	7
wms/ramsey	48	37	34	35	34	35	35	34
ms/cut/dimacs	62	48	4	4	4	4	4	4
wpms/aucsched	84	84	81	82	84	84	84	84
ms/bip-cut-140-630	100	83	0	0	0	0	0	0
wpms/min-enc/warehouses	18	2	1	2	18	18	18	18
pms/min-enc/kbtree	54	22	12	11	11	15	16	15
wpms/aucpaths	88	88	88	88	88	88	88	88
pms/csp/sparsetight	20	20	0	0	0	0	0	0
wpms/spot5log	21	4	6	6	6	6	6	6
pms/maxone/3sat	80	80	25	35	34	80	80	80
wms/cut/rand	40	40	0	0	0	0	0	0
wpms/QCP	25	20	25	25	25	25	25	25
pms/clique/struc	62	36	10	10	33	29	33	32
wms/cut/dimacs	62	55	3	3	3	3	3	3
ms/cut/rand	40	40	0	0	0	0	0	0
wpms/min-enc/planning	56	56	54	56	56	56	56	56
pms/frb	25	5	0	0	5	8	9	10
wpms/spot5dir	21	3	6	6	6	6	6	6
pms/csp/densetight	20	20	0	0	0	0	0	0
pms/pb/garden	7	5	5	5	5	6	6	6
Total	1960	1493	847	843	1042	1092	1085	1105

Table 4.4: Crafted instances: results for the best competing solver on Crafted instances (MiniSat), the overall best version of MaxHS from Chapter 3 (MaxHS-ch3 which is MaxHS-min-disj-inv-del), and five versions of MaxHS with non-core constraints and seeding. The table shows the number of instances solved in each benchmark family. For each family, the number of instances in the family is shown in column '#'. The name of the family begins with either 'ms', 'pms', 'wms' or 'wpms' which indicates whether or not the instances contain hard clauses ('p') and whether or not their soft clauses have non-uniform weights ('w'). The solvers are ordered by the total number of problems they solve.

Family	#	bincd	MaxHS ch3	MaxHS noncore imp	MaxHS noncore rev	MaxHS noncore	MaxHS noncore imp rev	MaxHS noncore eq
haplo-ped	100	23	46	29	29	28	29	28
pb-nencdr	128	116	109	106	106	103	106	104
pms/bcp-mtg	215	215	214	212	212	212	212	212
wpms/up-u98	80	79	80	80	80	80	80	80
ms/Safar	112	71	34	14	0	34	13	33
pms/pb/primes	86	76	74	76	79	76	80	80
pms/bcp-syn	74	45	67	65	71	66	71	71
pms/circtracecomp	4	4	0	0	0	0	0	0
pms/hap-asmbly	6	0	5	5	5	5	5	5
pb-nlogencdr	128	128	118	111	109	111	109	111
pms/bcp-fir	59	55	18	18	18	18	18	18
pms/pbo-rout	15	15	15	13	13	13	13	13
pms/pseudo/rout	15	15	15	15	15	15	15	15
aes	7	1	1	1	2	1	2	2
timetabling	32	12	6	7	8	8	7	8
pms/bcp-hipp	1183	1164	1140	1142	1141	1142	1141	1142
pms/pb/logic-syn	17	7	16	16	16	16	16	16
ms/circdebug	9	7	4	1	1	4	1	3
upgrade	100	97	100	100	100	100	100	100
pms/protein-ins	12	2	2	1	1	1	1	1
wpms/protein-ins	12	2	2	2	2	2	2	2
pms/bcp-msp	148	117	83	89	111	88	114	121
Total	2542	2251	2149	2103	2119	2123	2135	2165

Table 4.5: Industrial instances: results for the best competing solver on Industrial instances (BINCD), the overall best version of MAXHS from Chapter 3 (MAXHS-ch3 which is MAXHS-min-disj-inv-del), and five versions of MAXHS with non-core constraints and seeding. The table shows the number of instances solved in each benchmark family. For each family, the number of instances in the family is shown in column '#'. The name of the family begins with either 'ms', 'pms', 'wms' or 'wpms' which indicates whether or not the instances contain hard clauses ('p') and whether or not their soft clauses have non-uniform weights ('w'). The solvers are ordered by the total number of problems they solve.



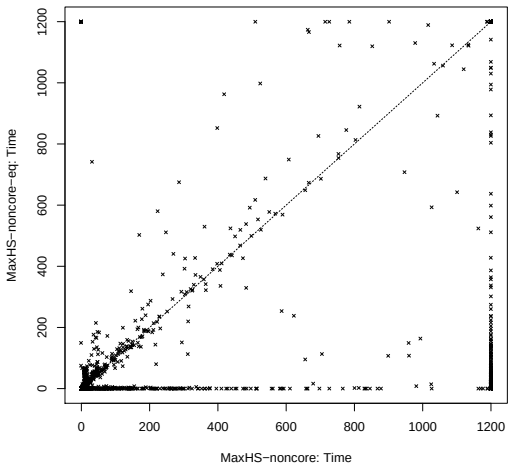
(a) Runtime

(b) LB Increment per CPLEX Call

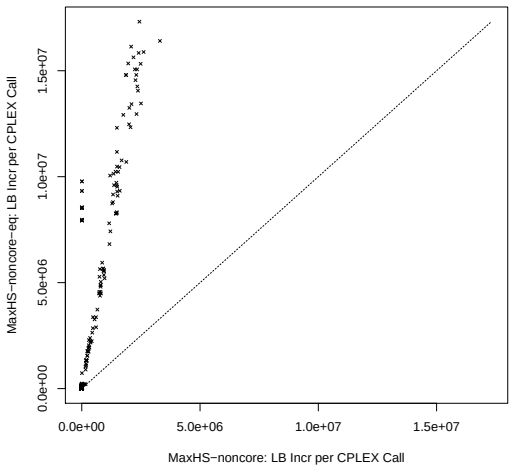
(c) SAT Time per CPLEX Call

(d) CPLEX Time per CPLEX Call

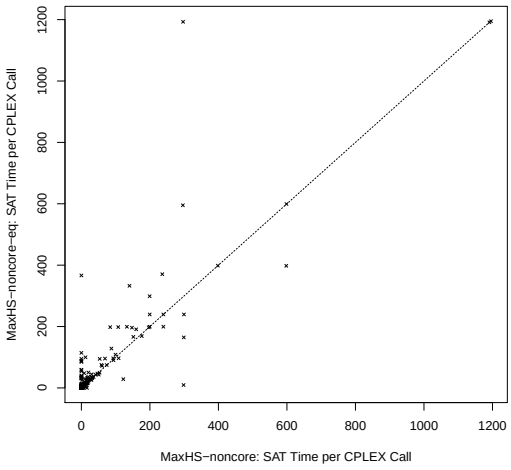
Figure 4.3: MAXHS-min-disj-inv-del vs. MAXHS-noncore



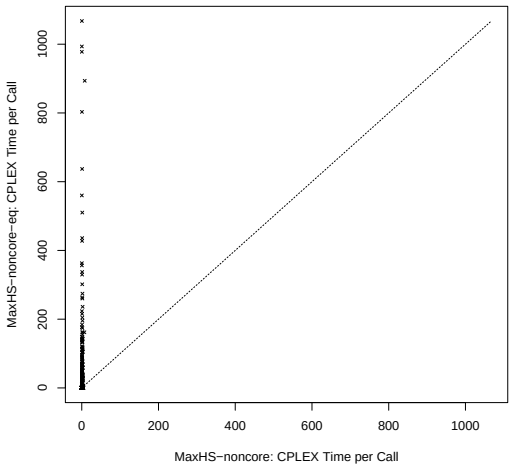
(a) Runtime



(b) LB Increment per CPLEX Call

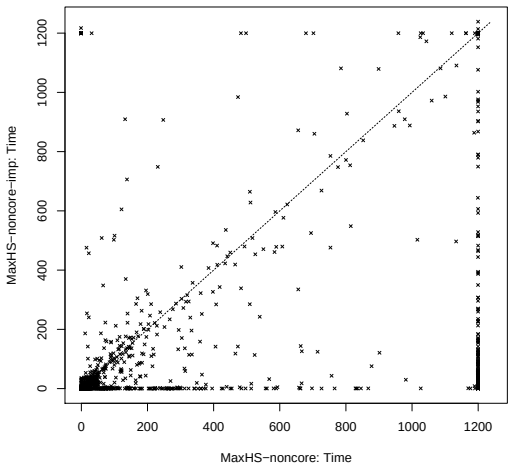


(c) SAT Time per CPLEX Call

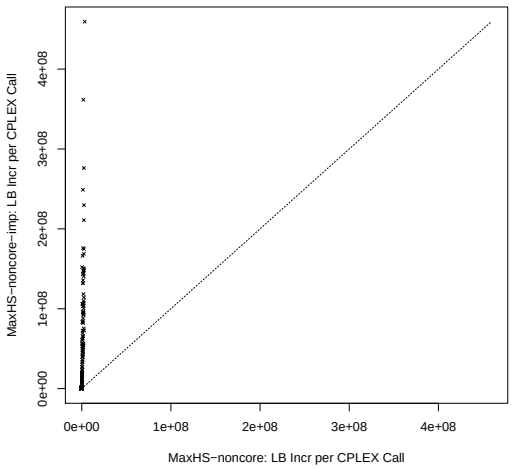


(d) CPLEX Time per CPLEX Call

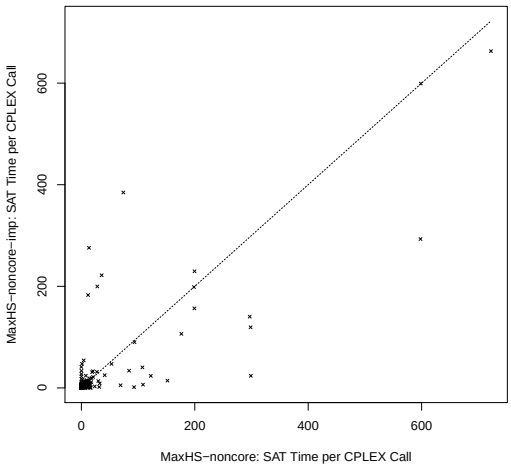
Figure 4.4: MAXHS-noncore vs. MAXHS-noncore-eq



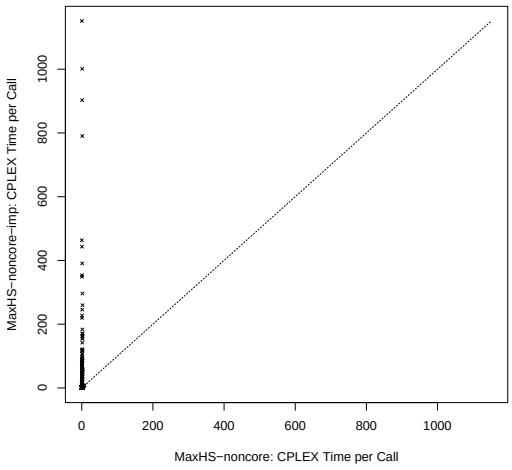
(a) Runtime



(b) LB Increment per CPLEX Call

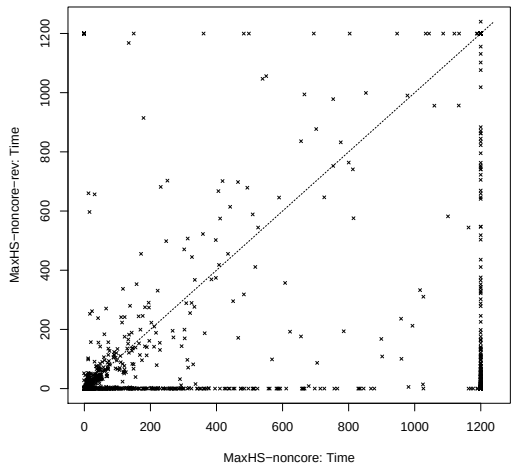


(c) SAT Time per CPLEX Call

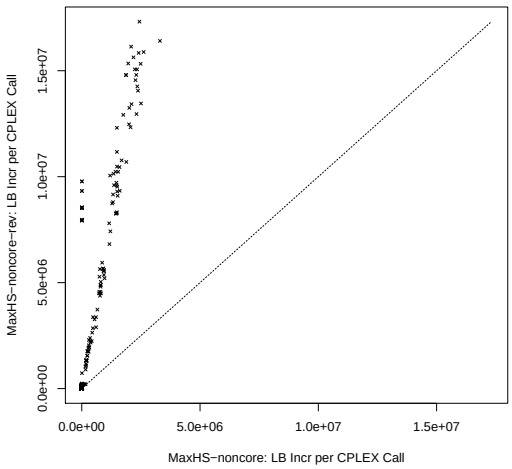


(d) CPLEX Time per CPLEX Call

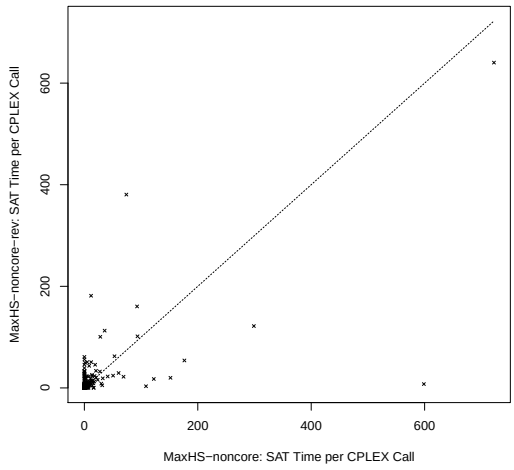
Figure 4.5: MAXHS-noncore vs. MAXHS-noncore-imp



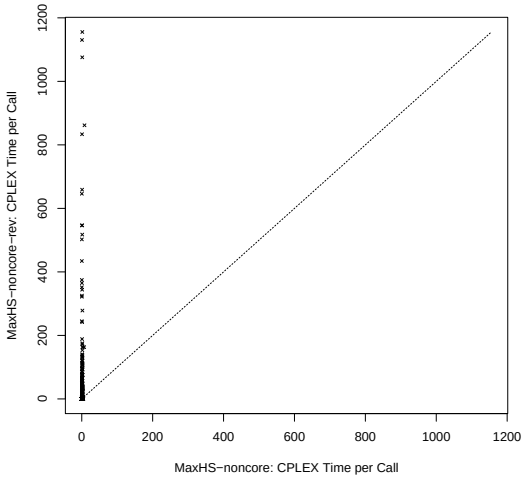
(a) Runtime



(b) LB Increment per CPLEX Call

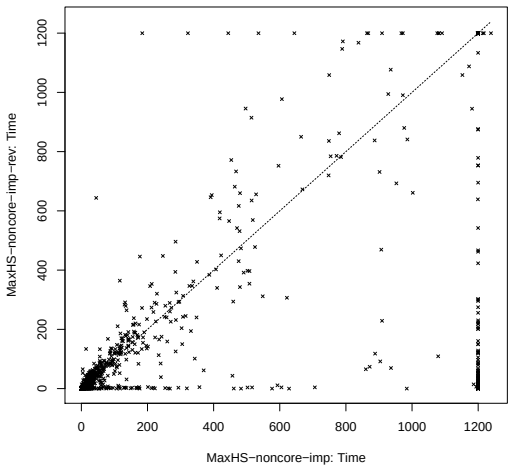


(c) SAT Time per CPLEX Call

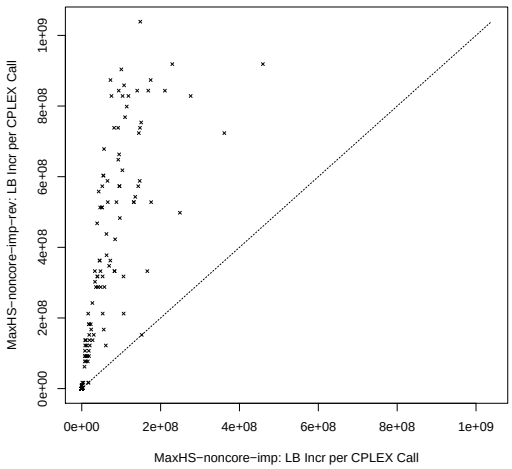


(d) CPLEX Time per CPLEX Call

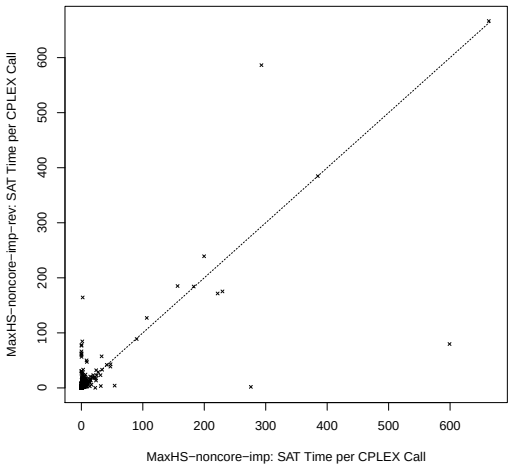
Figure 4.6: MAXHS-noncore vs. MAXHS-noncore-rev



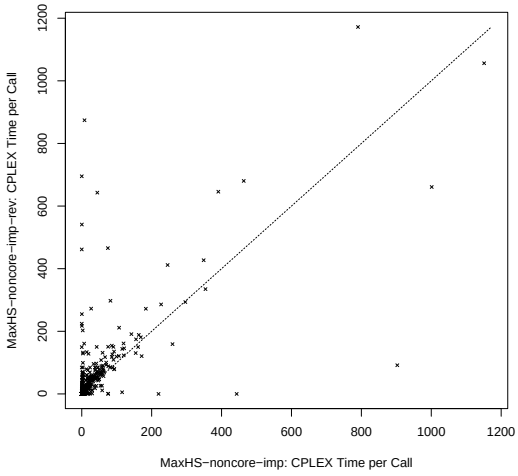
(a) Runtime



(b) LB Increment per CPLEX Call

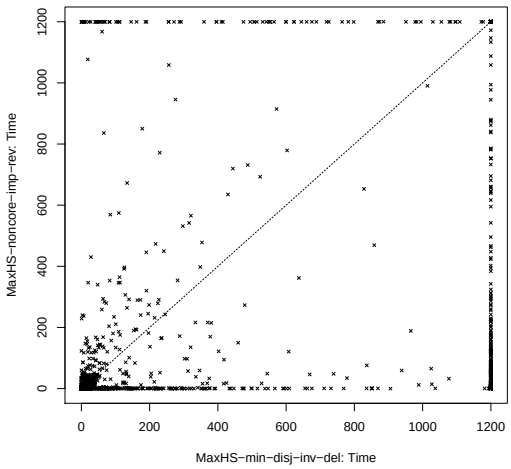


(c) SAT Time per CPLEX Call

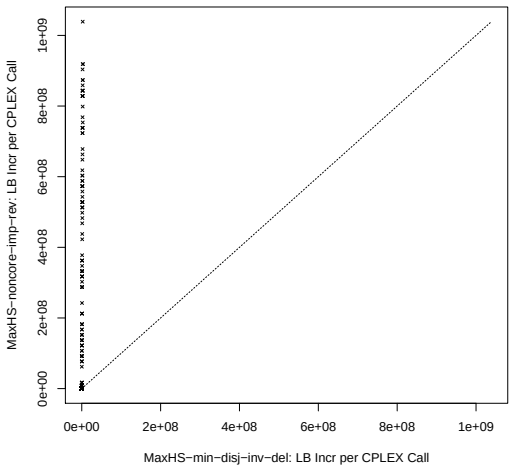


(d) CPLEX Time per CPLEX Call

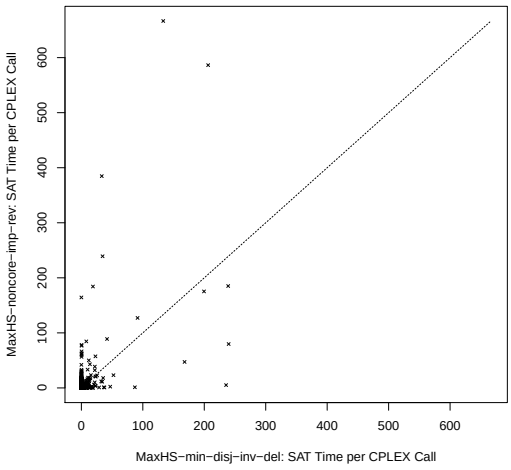
Figure 4.7: MAXHS-noncore-imp vs. MAXHS-noncore-imp-rev



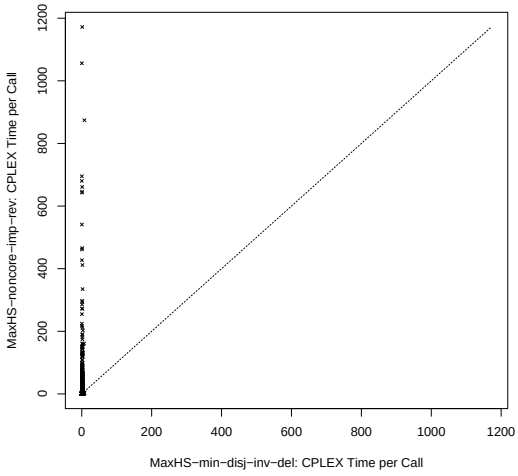
(a) Runtime



(b) LB Increment per CPLEX Call



(c) SAT Time per CPLEX Call



(d) CPLEX Time per CPLEX Call

Figure 4.8: MaxHS-min-disj-inv-del vs. MaxHS-noncore-imp-rev

Chapter 5

Non-Optimal Hitting Sets

5.1 Introduction

The MAXHS approach introduced in Chapter 3 involves solving many minimum cost hitting set problems, one after *every* new core is found. As the number of cores increases, the hitting set problems become larger and harder to solve. The minimum cost hitting set problem is itself an NP-hard optimization problem, so a natural question is whether or not performance can be improved by sometimes using non-optimal hitting sets that are cheaper to compute.

In this chapter, inspired by ideas originally proposed in the context of solving the Implicit Hitting set problem (Chandrasekaran, Karp, Moreno-Centeno, and Vempala, 2011b), we show how non-optimal hitting sets can be used in the MAXHS algorithm. The advantage of using non-optimal hitting sets is not only a reduction in the total time spent solving the hitting set problems. We see that non-optimal hitting sets also have a significant effect on which cores the SAT solver finds. Removing a set of clauses (i.e. the hitting set) from the MAXSAT formula means that the SAT solver is forced to find a refutation that does not use them. Using non-optimal hitting sets gives us more flexibility to control which and how many clauses are removed, and thus which cores are generated.

The goal is to improve the quality of the cores, in the sense that fewer cores are needed to find the MAXSAT solution.

5.2 MAXHS with Non-Optimal Hitting Sets

It is simple to modify the MAXHS algorithm to sometimes use non-optimal hitting sets, because the correctness of MAXHS relies mainly on two properties that are easily maintained. The first property is that MAXHS never generates the same core twice, since the next core is guaranteed to be un-hit by a hitting set of the previous cores. This property does not rely on the minimality of the hitting set. The second property is that at termination, the hitting set whose removal leaves a satisfiable formula must be minimal. Only the second property relies on the optimality of the hitting set. Therefore, we have a lot of flexibility to choose when non-optimal hitting sets should be used inside the MAXHS algorithm, as long as eventually the hitting set problem is solved to optimality to allow termination.

We first investigate how to use non-optimal hitting sets as much as possible in MAXHS, since solving the hitting set problems to optimality is quite expensive. That is, we propose that while the SAT solver is able to refute the remaining formula, we use non-optimal hitting sets. At some point, the SAT solver will no longer be able to find a refutation, because the non-optimal hitting set is large enough that it breaks all refutations. This is the point at which we solve the hitting set problem to optimality. Since the optimal hitting set is likely to be smaller, it may leave some cores un-hit. If so, we go back to generating cores using non-optimal hitting sets. Otherwise, the optimal hitting set hits all cores and the SAT solver will return the MAXSAT solution.

This new algorithm for MAXSAT is shown in Algorithm 9. We first describe its behaviour at a high level, relating it to the original MAXHS algorithm. Then, we describe the details and prove that it is correct.

Algorithm 9: An algorithm for solving MAXSAT that uses non-optimal hitting sets.

```
1  MAXSAT-solver-nonOPT (F)
2  (K, hs) = DisjointCores(F) /* hs is an optimal hitting set of K */
3  LB = cost(hs)
4  UB = ∞
5  π_UB = ∅ /* π_UB is the assignment that generated the current upper bound */
6  while true do
       // On return, hs hits all cores of F and F \ hs is satisfied by assignment π.
7      (corefound?, π) = GenerateCores(F, hs, K)
8      if cost(π) < UB then
9          UB = cost(π) /* Update the upper bound */
10         π_UB = π
11     if corefound? then
12         hs = FindMinCostHittingSet(K) /* Solve to optimality */
13         LB = cost(hs) /* Optimal hitting set gives a lower bound */
14     if LB == UB or not corefound? then
15         return (π_UB, cost(π_UB))
```

Every time through the main loop, non-optimal hitting sets are used to drive the discovery of cores (inside the function GenerateCores, see Algorithm 10). When no more cores can be found, GenerateCores returns and the algorithm will resort to calculating an optimal hitting set (line 12). The algorithm terminates when either the lower and upper bounds coincide, or no more cores can be found when the SAT solver is given an optimal hitting set (lines 14-15). In the latter case, correctness is by the same argument as for the original MAXHS algorithm. We show that the algorithm is correct in the former case as well, below.

We now describe the behaviour of Algorithm 9 in more detail. It maintains a collection of known cores $\mathcal{K}$, initialized with a set of disjoint cores and an optimal hitting set *hs* for them (line 2). Each iteration of the while loop on line 6 is called one *epoch*. In each epoch, the function GenerateCores is called with the current collection of cores $\mathcal{K}$ and a hitting set of $\mathcal{K}$. Note that every time GenerateCores is called, *hs* will be an *optimal* hitting set of $\mathcal{K}$, since the first time it is called the hitting set is optimal (by line 2) and every subsequent call only occurs if *corefound*? is *true*, in which case *hs* is optimal by

Algorithm 10: A function to generate cores using non-optimal hitting sets.

```
   // Precondition: hs is a hitting set of the set of cores 𝒦.
   // Postcondition: Returns core?=true iff a new core was added to 𝒦. ℱ \ hs is
      satisfied by κ.
 1 GenerateCores (ℱ, hs, 𝒦)
 2 core? = false
 3 while true do
 4 |   (sat?,κ) = SatSolver(ℱ \ hs)
   |   ; // If SAT, κ contains the satisfying truth assignment.
   |   ; // If UNSAT, κ contains a core.
 5 |   if sat? then
 6 |   |   break ; // Exit while loop
 7 |   core? = true
   |   // Add new core to set of cores
 8 |   𝒦 = 𝒦 ∪ {κ}
 9 |   hs = FindNonOptimalHittingSet(𝒦, κ, hs)
10 return (core?, κ)
```

line 12.

The function GenerateCores checks whether $\mathcal{F} \setminus hs$ is satisfiable by invoking a SAT solver (line 4), and if it is UNSAT, the SAT solver returns a new core κ that is added to $\mathcal{K}$ on line 8. We will consider various methods to calculate the hitting set for the current set of cores on line 9. GenerateCores continues to find and hit cores until hs is a hitting set of *all* cores of $\mathcal{F}$, at which point $\mathcal{F} \setminus hs$ is satisfiable so no new core can be found. The truth assignment satisfying $\mathcal{F} \setminus hs$ is then returned on line 10, along with the information whether or not any new core was added to $\mathcal{K}$. Note that if hs is not a minimum cost hitting set of $\mathcal{K}$, a satisfying assignment to $\mathcal{F} \setminus hs$ is not necessarily a MAXSAT solution.

When GenerateCores returns, if it has added another core to $\mathcal{K}$, the optimal hitting set is calculated (line 12 of Algorithm 9) and the lower bound is updated (line 13). Otherwise, GenerateCores has failed to find a new core, when passed an optimal hitting set, and the assignment last returned by the SAT solver is the MAXSAT solution by Theorem 3 on page 40. The MAXSAT solution has also been found if the best cost assignment returned by the SAT solver has cost equal to the lower bound. In this case, the proof that π_{UB} is

optimal is that the lower bound has equal cost.

We prove that Algorithm 9 is correct. We begin by proving that if a call to GenerateCores satisfies the preconditions then the postconditions will hold.

Proposition 12. *If hs is a hitting set of a set $\mathcal{K}$ of cores of* MAXSAT *instance $\mathcal{F}$, then Algorithm 10 invoked with input $(\mathcal{F}, hs, \mathcal{K})$ terminates. Upon return, $\mathcal{K}$, hs, and the two returned values (core?, κ) fulfill the following properties.*

1. *$\mathcal{K}$ is a possibly larger set of cores, containing all cores passed to Algorithm 10 as input.*

2. *core? is true if and only if Algorithm 10 added a new core of $\mathcal{F}$ to $\mathcal{K}$.*

3. *hs is a hitting set of $\mathcal{K}$.*

4. *κ is a truth assignment that satisfies $\mathcal{F} \setminus hs$.*

Proof. It is clear that at termination, $\mathcal{K}$ will contain all cores that were inputted to Algorithm 10, since $\mathcal{K}$ is only modified on line 8 by adding another core. Furthermore, $\mathcal{K}$ only contains cores of $\mathcal{F}$, since $\mathcal{K}$ is only augmented by κ's that have been returned by calls to SatSolver on line 4. It is also clear that *core*? will be *true* if and only if at least one core is added to $\mathcal{K}$, since whenever a core is added to $\mathcal{K}$ on line 8, the previously executed line set *core*? to *true*, and once it is set to *true* it can never be changed. If no cores are added to $\mathcal{K}$, then line 7 must never have been executed so *core*? will remain *false* as it was initialized on line 2.

Next, we show that *hs* is always a hitting set of $\mathcal{K}$. This is clear from lines 8 and 9, since every time $\mathcal{K}$ is augmented with a new core, the value of *hs* is immediately updated to be a hitting set of $\mathcal{K}$. These are the only lines that modify $\mathcal{K}$ and *hs*.

If the algorithm GenerateCores terminates, it must be the case that the last call to SatSolver returned *sat*? = *true*. In this case, the SAT solver also returns κ equal to a truth assignment satisfying $\mathcal{F} \setminus hs$, and this value of κ is returned on line 10 as desired.

Finally, we show that the algorithm terminates. Algorithm 10 terminates if the call to SatSolver on line 4 returns $sat? = true$, that is, if $\mathcal{F} \setminus hs$ is satisfiable. We argue that this must eventually be the case, by showing that every call to SatSolver that returns $sat? = false$ generates a distinct core. Thus, since there are a finite number of different cores of $\mathcal{F}$, eventually SatSolver must be unable to find another core. To see that every call to SatSolver must generate a distinct new core, note that if κ is a core returned by SatSolver, $\kappa \subseteq \mathcal{F} \setminus hs$. However, we have already shown that hs is a hitting set of all previously found cores. So since κ is not hit by hs, κ can not be the same as any core in $\mathcal{K}$.

□

Proposition 13. *Algorithm 9 correctly returns a solution to the inputted* MAXSAT *problem $\mathcal{F}$. That is, it returns a truth assignment π for $\mathcal{F}$ that achieves $mincost(\mathcal{F})$.*

Proof. The algorithm only terminates on line 15, and there are two cases depending on how the condition on line 14 was satisfied.

The first case is when line 15 is executed because the condition $LB == UB$ was satisfied. However, line 15 can only be reached if GenerateCores has been called at least once. We argue that the preconditions of GenerateCores hold every time it is called. The first time GenerateCores is called, $\mathcal{K}$ and hs were produced by DisjointCores (see Algorithm 6 on page 54) so they satisfy the preconditions. On subsequent calls to GenerateCores, lines 12-13 must have been executed right before GenerateCores was called, and therefore hs is a MCHS of $\mathcal{K}$ by the correctness of function FindMinCostHittingSet. Therefore, by the correctness of GenerateCores (Proposition 12), π_{UB} must have been assigned on line 10 to a truth assignment with finite cost equal to UB.[1] Furthermore, the value of LB is always a correct lower bound on $mincost(\mathcal{F})$, since it is only updated on lines 3 and 13 where by Proposition 3 on page 39 it is a valid lower bound. Therefore,

[1]The truth assignment π returned by GenerateCores will always have finite cost, because it satisfies $\mathcal{F} \setminus hs$ where $hs \subseteq soft(\mathcal{F})$ and therefore $hard(\mathcal{F}) \subseteq \mathcal{F} \setminus hs$ so every clause with infinite weight is satisfied by π.

when π_{UB} is returned on line 15 it is a MAXSAT solution.

The second case is when line 15 is executed because *corefound*? was false. In this case, GenerateCores did not find an un-hit core in $\mathcal{F} \setminus hs$. Every time GenerateCores is called, hs is a MCHS of $\mathcal{K}$ so if $\mathcal{F} \setminus hs$ is satisfiable, by Theorem 3 on page 40, the truth assignment π returned by the last call to GenerateCores is a MAXSAT solution. There is no smaller cost truth assignment, so $cost(\pi_{UB}) \geq cost(\pi)$. Therefore, line 15 returns a MAXSAT solution in this case as well.

Finally, we must argue that line 15 is eventually executed. It will be executed if *corefound*? is false, so suppose *corefound*? is always true. Therefore, every call to GenerateCores finds a core. This core must be distinct from every previously found core in $\mathcal{K}$, since it is a subset of $\mathcal{F} \setminus hs$ (line 4 of GenerateCores) so it is not hit by hs, a hitting set of $\mathcal{K}$ (by the precondition and line 9 of GenerateCores). But there are only a finite number of cores of $\mathcal{F}$, so this is a contradiction. Therefore, eventually *corefound*? will be false and the program will terminate. □

5.2.1 Methods to Compute Non-Optimal Hitting Sets

It remains to specify how the non-optimal hitting sets are calculated (function FindNonOptimalHittingSet). The least computationally expensive method is to augment the current hitting set, which is passed as the second argument to FindNonOptimalHittingSet. The existing hitting set can be augmented by, e.g., adding any clause chosen randomly from the newest core κ. Or, rather than choosing a clause at random, we can take a clause that appears in the maximum number of cores in $\mathcal{K}$. We found this heuristic improved performance on a small test set of instances. A possible explanation for the good behaviour of this heuristic is that clauses that the SAT solver prefers to use in its refutations are quickly added to the hitting set. Therefore, these clauses are no longer available to the SAT solver, and they will not appear in any more cores found during the current epoch. This encourages the SAT solver to refute the remaining theory in a

different way than before. It is also possible to add more than one clause to the hitting set at a time. The best performing heuristic, found by trial and error, is to take the top 10% of the newest core's clauses (sorted by their occurrences in $\mathcal{K}$) and add them to the non-optimal hitting set.

We also consider re-computing a non-optimal hitting set from scratch when FindNonOptimalHittingSet is called. In this case, FindNonOptimalHittingSet ignores the current hitting set passed to it, and instead applies a well-known greedy algorithm for the MCHS problem (Johnson, 1973). The greedy algorithm simply builds a hitting set by choosing the remaining clause that hits the largest number of cores for the smallest cost, i.e. the clause c that minimizes $\frac{|\{\kappa \in \mathcal{K} : c \in \kappa\}|}{wt(c)}$. As clauses are added to the greedy hitting set hs, the working problem $\mathcal{K}$ is simplified by removing all cores that are hit.

5.3 Combining Seeding and Non-Optimal Hitting Sets

In Chapter 4 we saw that the technique of seeding the MIP solver with constraints is very effective in improving the robustness of MAXHS. In this section we show that the MAXHS algorithm that uses non-optimal hitting sets, Algorithm 9, can be enhanced by also using seeding.

Several different methods of identifying constraints with which to seed the MIP solver were discussed in Section 4.2.3 on page 91. The overall most effective method of seeding was observed to be Equivalence seeding. This type of seeding is also the most easily combined with Algorithm 9, because it only requires access to $\mathcal{F}^b$ and not $\mathcal{F}^b_{eq}$. This is because Equivalence seeding only needs to know which b-variables relax soft unit clauses of $\mathcal{F}$.

In order to combine Equivalence seeding with Algorithm 9, we only need to modify two lines of Algorithm 9. First, we perform Equivalence seeding after the disjoint core phase. Second, on line 12, instead of finding a minimal cost hitting set of the known

cores, we find a minimal cost hitting set of the known cores that satisfies these extra seeded constraints.

It is easy to see that with these changes Algorithm 9 is still correct, by observing that any clause added to the MIP model by Equivalence seeding is implied by $\mathcal{F}^b$ (as well as $\mathcal{F}^b_{eq}$) and then applying an argument similar to the proof of Theorem 4 for the Realizability condition.

5.4 Experimental Evaluation

In this section we examine the empirical behaviour of Algorithm 9 and the various methods of generating non-optimal hitting sets described in Section 5.2.1, as well as the combination of Equivalence seeding and non-optimal hitting sets from Section 5.3. The experimental setup is the same as in Section 3.7. All of the techniques introduced in this chapter were implemented on top of the best version of MAXHS from Chapter 3, MAXHS-min-disj-inv-del. Therefore, we will omit the designations "min-disj-inv-del" when naming the versions of MAXHS that include techniques from the current chapter. We report results with four versions of MAXHS that use Algorithm 9, listed in Table 5.1.

MAXHS Version	Equivalence Seeding	Non-Opt HS (Alg. 9)	maxoccur HS	10percent HS	greedy HS
MAXHS-nonopt-maxoccur		✓	✓		
MAXHS-nonopt-10percent		✓		✓	
MAXHS-nonopt-greedy		✓			✓
MAXHS-nonopt-10percent-eqseed	✓	✓		✓	

Table 5.1: The four versions of MAXHS that we evaluate in this chapter.

Overall Performance

We first present the overall performance of MAXHS using non-optimal hitting sets using the two cactus plots in Figures 5.1 and 5.2. The performance of the best version, MAXHS-nonopt-10percent-eqseed, surpasses the performance of the best MAXHS solvers from the previous two chapters and is clearly the most robust solver for MAXSAT. We observe that

the particular method of building the non-optimal hitting sets does not have a significant effect on overall performance, as shown in Figure 5.2.

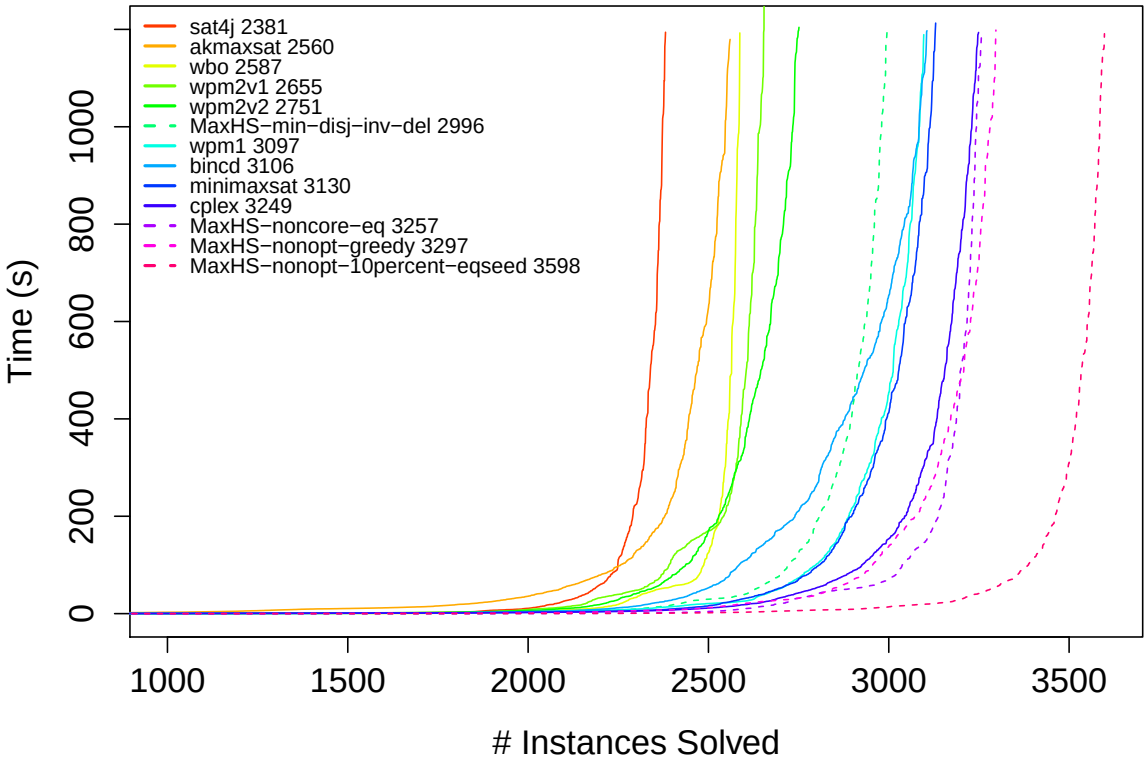


Figure 5.1: Runtime results for the competing solvers, the overall best version of MaxHS from Chapter 3 (MaxHS-min-disj-inv-del) and Chapter 4 (MaxHS-noncore-eq), the overall best version of MaxHS with non-optimal hitting sets (MaxHS-nonopt-greedy) and MaxHS with Equivalence seeding and non-optimal hitting sets (MaxHS-nonopt-10percent-eqseed). Shows how many problems were solved within each time limit. The total number of instances solved is given in the legend after the solver's name.

Tables 5.2 and 5.3 show the number of problems each version solved, broken down by benchmark family and divided into Crafted and Industrial instances according to the MaxSAT Evaluation categorization. We observe that using non-optimal hitting sets benefits both kinds of instances, Crafted and Industrial. This is interesting because three of these versions of MaxHS do not seed CPLEX with constraints. So this demonstrates that two very different techniques (non-optimal hitting sets and seeding CPLEX) can

both achieve the desired result of a solver that is robust on both Crafted and Industrial problems.

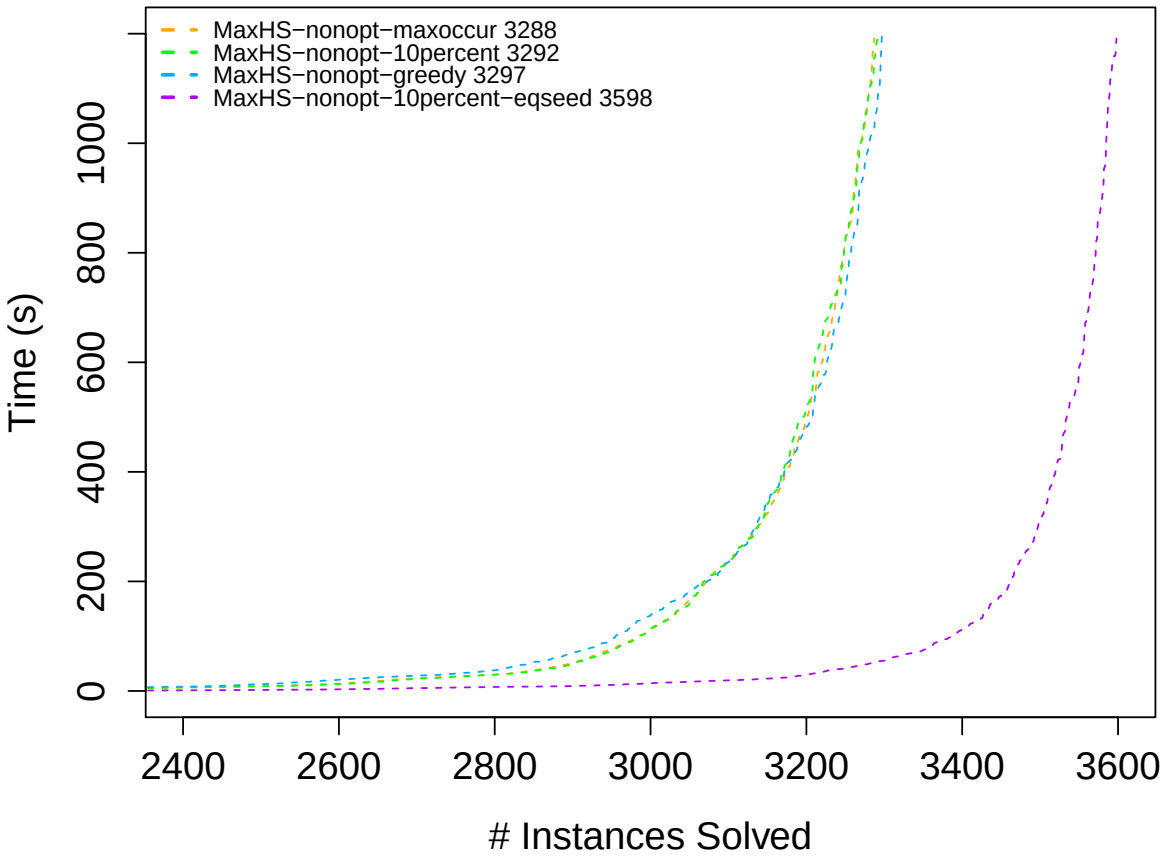


Figure 5.2: Runtime results for four versions of MaxHS with non-optimal hitting sets. Shows how many problems were solved within each time limit. The total number of instances solved is given in the legend after the solver's name.

Trade-offs in MaxHS

Next, we study the behaviour of MaxHS with non-optimal hitting sets in greater depth. As described in Section 3.7.2, we collected statistics from the runs of each version of our solver on each instance, and in Figures 5.3-5.5, we compare versions of MaxHS along four dimensions: the runtime on each instance, the CPLEX time per CPLEX call, the SAT time per CPLEX call, and the increment in the cost of the lower bound per CPLEX call.

All 4502 instances appear in each of these graphs, although the number of data points sometimes appears to be fewer because they overlap.

We observe that using non-optimal hitting sets tends to increase the time spent in the SAT solver, per call to CPLEX. This makes sense since between every call to CPLEX, many cores are generated by the SAT solver, in a process that is guided by the non-optimal hitting sets. We also see that the increment in the lower bound per CPLEX call is much higher when using non-optimal hitting sets (see Figures 5.3-5.5(b)). This effect is similar to what we observed in the case of seeding CPLEX with non-core constraints and also the disjoint cores phase (compare the graphs in this section to those in Sections 3.7.2 and 4.2.5). The main benefit of using non-optimal hitting sets is that by using more SAT solving time to find cores, we are able to supply CPLEX with more information. This rules out many low-cost assignments that CPLEX would otherwise return, thus reducing the total number of calls to CPLEX required. And again, we see that although the extra constraints increase the size of the problems CPLEX has to solve, any increase in time per CPLEX call is usually offset by a reduction in the total number of CPLEX calls.

5.5 Related Work

Chandrasekaran et al. proposed an algorithm to solve the Implicit Hitting Set problem (defined in Section 3.8) that is closely related to Algorithm 9 (Chandrasekaran et al., 2011b). They first considered using an approach like our Algorithm 5, but they were able to obtain much better performance by using non-optimal hitting sets to build up the collection of known sets to hit. Their experience with solving the IHS problem motivated us to try non-optimal hitting sets in MAXHS.

5.6 Conclusion

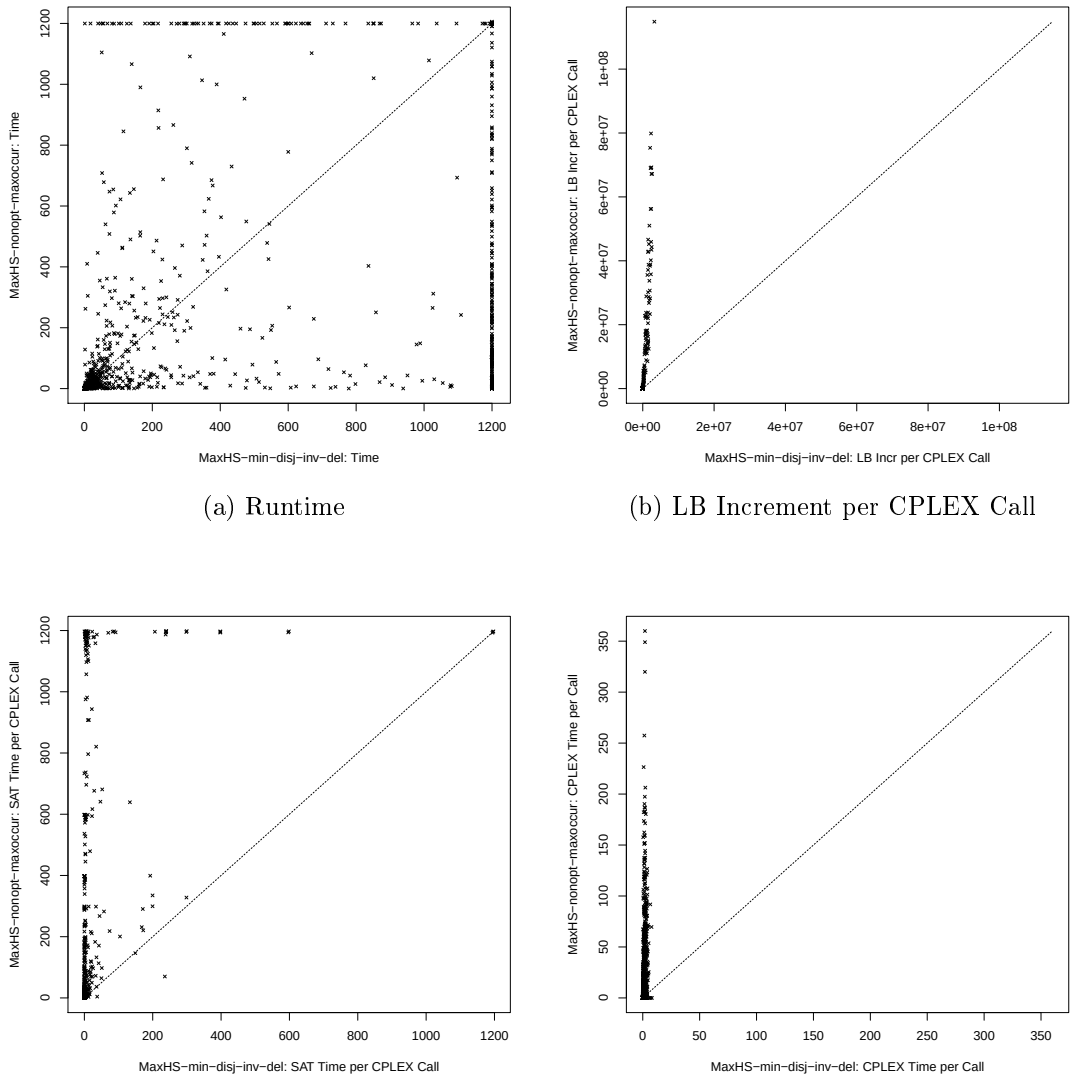
In this chapter we proposed to use non-optimal hitting sets in the maxhs approach introduced in Chapter 3. The non-optimal hitting sets require much less time to compute than cplex takes to solve the problems optimally, however, in order to achieve a complete algorithm we must eventually solve a hitting set problem to optimality. We observed that the effect of using non-optimal hitting sets is that a larger number of cores is collected between each expensive call to cplex. Thus, like non-core constraints and seeding in Chapter 4, by providing more information to cplex we can reduce the total number of MCHS problems solved. The resulting solver, maxhs-nonopt-10percent-eqseed, is robust across Crafted and Industrial instances and solves many more problems overall than any other version of maxhs, cplex itself, and the competing solvers including minimaxsat and bincd.

Family	#	mini	maxhs ch3	maxhs ch4	maxhs nonopt 10percent	maxhs nonopt maxoccur	maxhs nonopt greedy	maxhs nonopt 10percent eqseed
ms/spinglass	20	20	0	0	0	1	4	1
wms/kexu/frb-wcnf	35	15	10	20	16	16	15	20
pms/csp/sparseloose	20	20	20	11	20	20	20	20
wpms/pb/factor/	186	186	186	186	186	186	186	186
pms/csp/denseloose	20	20	10	0	13	15	15	14
KnotPipatsrisawat	350	117	61	52	286	290	260	284
pms/queens	7	7	5	3	5	5	4	5
wpms/aucregions	84	84	35	84	4	4	13	84
ms/cut/spinglass	5	3	1	1	1	1	2	1
pms/jobshop	4	2	4	3	4	4	4	4
wpms/planning	71	71	69	71	71	71	71	71
pms/maxone/struc	60	60	46	60	55	54	57	60
ms/ramsey	48	35	34	34	34	34	34	34
pms/clique/rand	96	96	4	96	4	4	44	96
wms/cut/spinglass	5	4	1	1	1	1	2	1
wpms/pb/miplib	16	5	7	7	7	7	7	7
wms/ramsey	48	37	34	35	34	34	34	34
ms/cut/dimacs	62	48	4	4	4	4	4	4
wpms/aucsched	84	84	81	84	76	76	77	84
ms/bip-cut-140-630	100	83	0	0	0	0	0	0
wpms/min-enc/warehouses	18	2	1	18	8	9	1	18
pms/min-enc/kbtree	54	22	12	15	13	12	12	19
wpms/aucpaths	88	88	88	88	88	88	88	88
pms/csp/sparsetight	20	20	0	0	12	15	12	11
wpms/spot5log	21	4	6	6	6	6	6	6
pms/maxone/3sat	80	80	25	80	45	45	44	80
wms/cut/rand	40	40	0	0	0	0	0	0
wpms/QCP	25	20	25	25	25	25	25	25
pms/clique/struc	62	36	10	29	12	12	17	34
wms/cut/dimacs	62	55	3	3	4	4	5	4
ms/cut/rand	40	40	0	0	0	0	0	0
wpms/min-enc/planning	56	56	54	56	56	56	56	56
pms/frb	25	5	0	8	5	5	0	9
wpms/spot5dir	21	3	6	6	6	6	6	6
pms/csp/densetight	20	20	0	0	1	3	6	2
pms/pb/garden	7	5	5	6	5	5	6	6
Total	1960	1493	847	1092	1107	1118	1137	1374

Table 5.2: Crafted instances: results for the best competing solver on Crafted instances (minimaxsat), the overall best versions of maxhs from Chapter 3 (maxhs-ch3 which is maxhs-min-disj-inv-del) and Chapter 4 (maxhs-ch4 which is a re-naming of maxhs-noncore-eq), and four versions of maxhs with non-optimal hitting sets.The table shows the number of instances solved in each benchmark family. For each family, the number of instances in the family is shown in column '#'. The name of the family begins with either 'ms', 'pms', 'wms' or 'wpms' which indicates whether or not the instances contain hard clauses ('p') and whether or not their soft clauses have non-uniform weights ('w'). The solvers are ordered by the total number of problems they solve.

Family	#	bincd	MAXHS ch3	MAXHS ch4	MAXHS nonopt greedy	MAXHS nonopt maxoccur	MAXHS nonopt 10percent	MAXHS nonopt 10percent eqseed
haplo-ped	100	23	46	28	33	26	31	32
pb-nencdr	128	116	109	104	128	118	116	117
pms/bcp-mtg	215	215	214	212	214	215	215	215
wpms/up-u98	80	79	80	80	80	80	80	80
ms/Safar	112	71	34	33	34	33	36	34
pms/pb/primes	86	76	74	80	74	77	77	80
pms/bcp-syn	74	45	67	71	69	69	69	71
pms/circtracecomp	4	4	0	0	1	0	0	0
pms/hap-asmbly	6	0	5	5	5	5	5	5
pb-nlogencdr	128	128	118	111	128	128	128	128
pms/bcp-fir	59	55	18	18	18	32	40	40
pms/pbo-rout	15	15	15	13	10	12	11	12
pms/pseudo/rout	15	15	15	15	11	11	11	13
aes	7	1	1	2	2	2	2	2
timetabling	32	12	6	8	7	7	7	7
pms/bcp-hipp	1183	1164	1140	1142	1138	1141	1143	1141
pms/pb/logic-syn	17	7	16	16	16	16	16	16
ms/circdebug	9	7	4	3	3	4	4	4
upgrade	100	97	100	100	100	100	100	100
pms/protein-ins	12	2	2	1	1	2	2	2
wpms/protein-ins	12	2	2	2	1	2	2	2
pms/bcp-msp	148	117	83	121	87	90	90	123
Total	2542	2251	2149	2165	2160	2170	2185	2224

Table 5.3: Industrial instances: results for the best competing solver on Industrial instances (BINCD), the overall best versions of MAXHS from Chapter 3 (MAXHS-ch3 which is MAXHS-min-disj-inv-del) and Chapter 4 (MAXHS-ch4 which is a re-naming of MAXHS-noncore-eq), and four versions of MAXHS with non-optimal hitting sets. The table shows the number of instances solved in each benchmark family. For each family, the number of instances in the family is shown in column '#'. The name of the family begins with either 'ms', 'pms', 'wms' or 'wpms' which indicates whether or not the instances contain hard clauses ('p') and whether or not their soft clauses have non-uniform weights ('w'). The solvers are ordered by the total number of problems they solve.



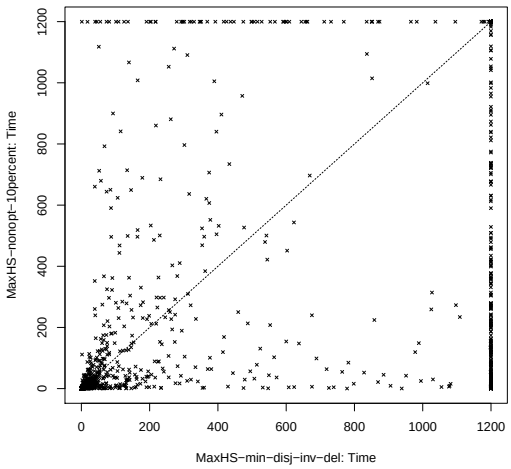
(a) Runtime

(b) LB Increment per CPLEX Call

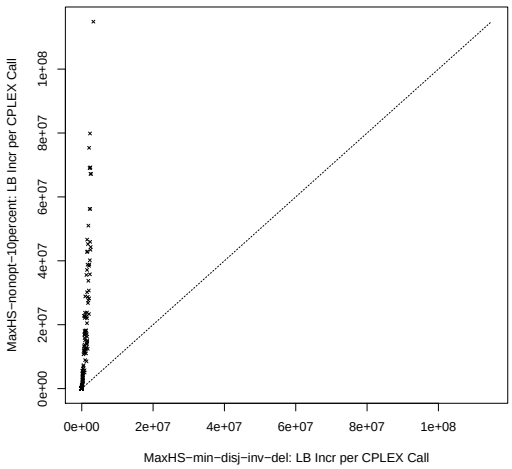
(c) SAT Time per CPLEX Call

(d) CPLEX Time per CPLEX Call

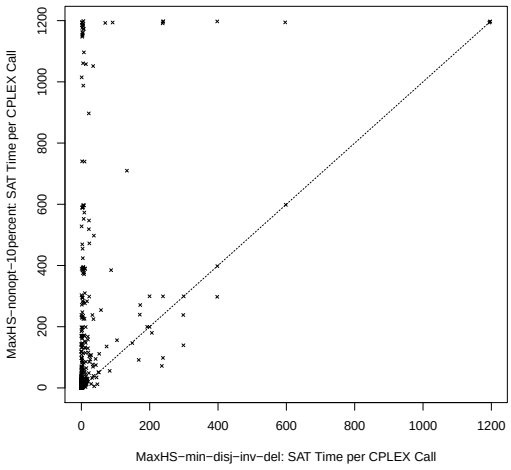
Figure 5.3: MaxHS-min-disj-inv-del vs. MaxHS-nonopt-maxoccur



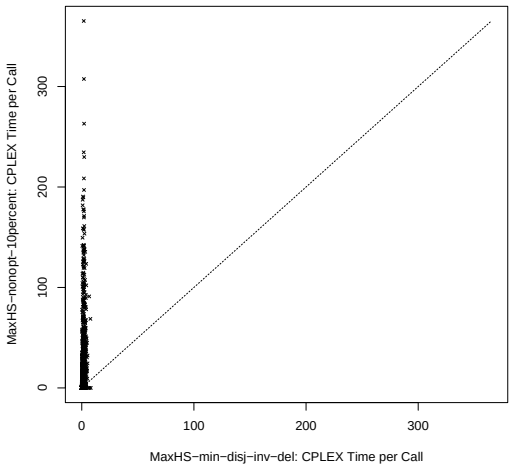
(a) Runtime



(b) LB Increment per CPLEX Call

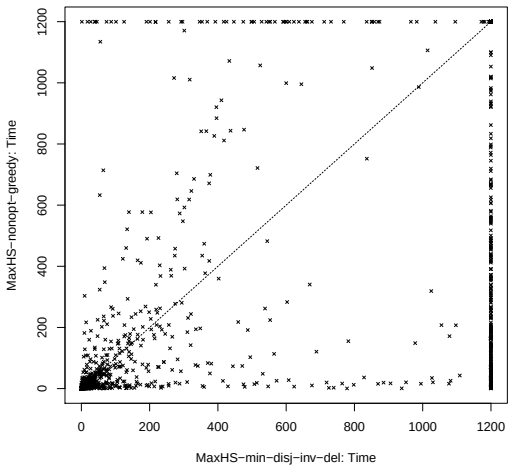


(c) SAT Time per CPLEX Call

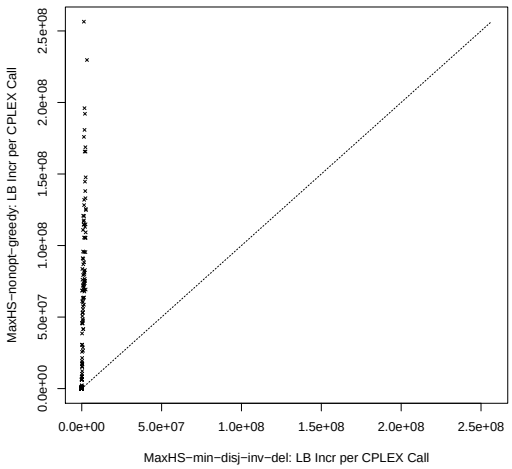


(d) CPLEX Time per CPLEX Call

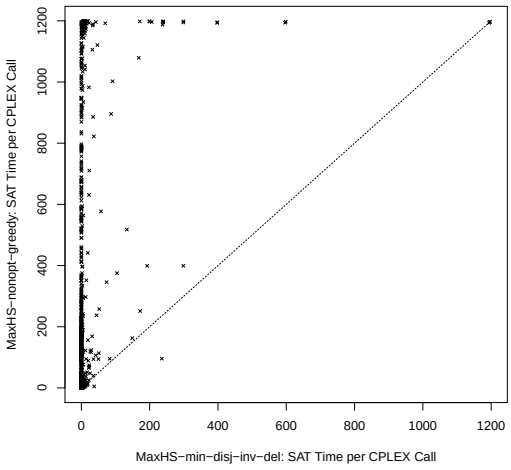
Figure 5.4: MaxHS-min-disj-inv-del vs. MaxHS-nonopt-10percent



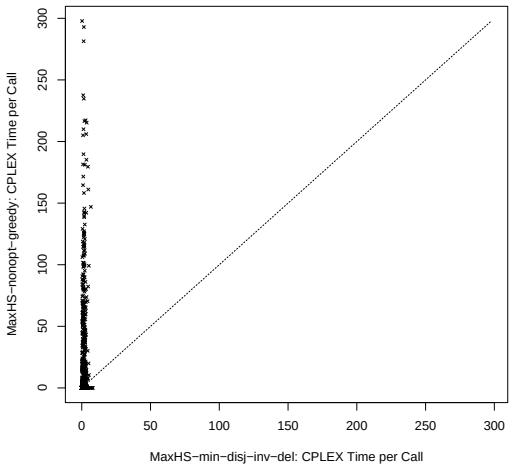
(a) Runtime



(b) LB Increment per CPLEX Call



(c) SAT Time per CPLEX Call



(d) CPLEX Time per CPLEX Call

Figure 5.5: MaxHS-min-disj-inv-del vs. MaxHS-nonopt-greedy

Chapter 6

Hitting Set Bounds in Branch and Bound for MAXSAT

6.1 Introduction

In Chapter 2, we described the two main approaches for solving MAXSAT: exploiting a SAT solver to solve a sequence of SAT problems, and Branch and Bound search. In Chapters 3– 5, we proposed a new family of algorithms that are similar to the former class because they also use a SAT solver to solve a sequence of SAT problems. In this chapter we turn our attention to developing a new Branch and Bound algorithm for MAXSAT. We first provide evidence that a Branch and Bound approach is better suited to particular MAXSAT instances than MAXHS or any other SAT-based MAXSAT solver. This motivates us to consider how to apply the ideas introduced in the previous chapters in order to create a robust Branch and Bound solver for MAXSAT.

We show how conflicts in the reduced MAXSAT formula can be learned at each node of the Branch and Bound search. These conflicts can be thought of as context-dependent cores. Thus we propose to use them in a MCHS approach, to generate lower bounds and prune the search space. In contrast to existing Branch and Bound solvers for MAXSAT,

our algorithm performs *learning* in the same spirit as clause learning in DPLL. The conflict information we discover in one part of the search tree can be saved and used again anywhere else it is relevant.

Additionally, since MCHS is itself an NP-hard problem, some tractable techniques for computing lower bounds on the exact solution are introduced. Two new greedy heuristics for hitting set are given, one of which improves on the heuristic in (Petit, Bessière, and Régin, 2003). A third alternative is to formulate the minimal hitting set problem as an integer program and use linear programming to provide a lower bound approximation.

6.2 Branch and Bound vs. Sequence of SAT

The maxhs algorithm works by finding cores of the maxsat formula and solving a MCHS problem over them. Thus, as explained in Section 3.4, the efficiency of maxhs is influenced by three main factors: the difficulty of refuting the maxsat formula to find the cores, the total number of cores that must be found, and the structure of the hitting set problems produced by the cores. Similarly, the behaviour of other sequence of SAT algorithms also depends mostly on the difficulty of refuting the original maxsat formula, and the properties of the cores such as their size and how much they overlap.[1] On the other hand, sequence of SAT approaches, including maxhs, are highly suited to instances whose cores are quite disjoint and much smaller in size relative to the size of the maxsat instance. This kind of structure is often found in maxsat instances arising from industrial applications.

In contrast, the performance of current Branch and Bound algorithms for maxsat appears to be highly dependent on the size of the maxsat instance and the length of its clauses. The key contributions in existing work are various techniques for computing good lower bounds during search. Most of these techniques can be understood as applying

[1] For example, if the cores are large, then the number of relaxation variables will also be large, even if they are only added to clauses that appear in found cores. If the cores also share many clauses, then the technique of organizing the discovered cores into disjoint covers, as done by wpm2, will be ineffective.

incomplete restrictions of MAXRES inference (Larrosa et al., 2008). Since MAXRES can inflate the size of the MAXSAT formula (see Section 2.4), previous work has concentrated on finding restricted cases where MAXRES inferences can be more efficiently applied. These cases usually apply only to clauses of length three or less; longer clauses do not contribute to the lower bound. Therefore, instances with many long clauses will have very deep search trees since the long clauses will not help to prune the search until almost all of their literals have been falsified. Furthermore, current Branch and Bound solvers look for these restricted cases in the *reduced* MAXSAT theory. As a result, all of the inferences must be undone on backtrack and recomputed from scratch along future branches of the search tree. Since the search can not learn as it progresses, the strength of the pruning is limited. This means that on large MAXSAT instances or ones with long clauses, Branch and Bound becomes ineffective. However, if the MAXSAT instance is not too large and its clauses are short, a Branch and Bound solver may be able to solve it, even though all SAT-based methods fail due to unfavourable core structure.

We therefore propose a new Branch and Bound algorithm that attempts to address the current weaknesses mentioned above. We introduce a new lower bound that like previous bounds, uses unit propagation to efficiently identify conflicts. However, our lower bound will be applicable to all such conflicts, regardless of the number of clauses involved and their lengths. Furthermore, we save these identified conflicts so they can be used again in other regions of the search space, even after backtrack. This allows the bounds to be strengthened by learning as the search progresses. The result is a much stronger method of pruning the Branch and Bound search that can significantly reduce the total number of nodes searched. However, questions arise as to how to find the most useful conflicts, when to save them, and how to use these conflicts to compute a lower bound that is effective but cheap enough to apply at every node.

6.3 Hitting Set Bounds

At each node n of the Branch and Bound search, we wish to calculate a lower bound on $mincost(n)$, the cost of the minimal cost complete assignment below that node.

Definition 19 ($mincost(n)$ for a node n)**.** *Let n be a node in a Branch and Bound search and let π be the assignments made at n. Then $mincost(n)$ is the cost of the minimal cost complete truth assignment extending π.*

The remaining formula at a node of Branch and Bound is formally defined as the reduction of the original formula by the assignments at that node.

Definition 20 (Reduced maxsat Instance)**.** *If $\mathcal{F}$ is a* maxsat *instance, n is a node of Branch and Bound search, and π is the partial truth assignment at n, let $\mathcal{F}|_n$ be the reduction of $\mathcal{F}$ by π, where every clause satisfied by π has been removed, all literals falsified by π have been removed from the remaining clauses, and the clause weights remain the same.*

Proposition 14. *The cost of a solution to the* maxsat *instance $\mathcal{F}|_n$ is equal to $mincost(n)$.*

Proof. Observe that for every clause of $\mathcal{F}$ falsified at n, $\mathcal{F}|_n$ contains an empty clause of equal weight. □

A common approach in maxsat is to find ways to anticipate that some clauses of the remaining formula will be falsified. For example, if the remaining formula contains two conflicting unit clauses (x) and $(\neg x)$, exactly one of these two clauses will be falsified at every leaf in the subtree. More generally, if we know that at least one out of a set of clauses κ will be falsified by every truth assignment extending the current node, then this is evidence that at least the cost of the minimum cost clause in κ must be incurred.

Such a set of clauses as κ can be thought of as a core of the remaining formula. This suggests that if we have more than one such set of clauses, their MCHS should provide a lower bound on the cost of a solution to the remaining formula. This is an alternative to

the technique of dealing with overlapping conflicts by transformation, used by existing MAXSAT solvers. The advantage of the MCHS bound is that it is suited to handling arbitrary conflicts, while existing MAXSAT solvers only exploit simple conflicts so as to avoid transformations that increase the size of the remaining formula.

However, our goal is not only to deal with overlapping conflicts effectively. We also wish to learn information during search that can be used elsewhere in search. If we just look for conflicts in the remaining formula at a node and combine overlapping conflicts using a MCHS computation, we will lose all of this work when we backtrack. The key insight is that identifying a core/conflict in the remaining formula actually corresponds to learning a clause (from the original MAXSAT formula) that is falsified at the current node.

The following observation says that any conflict in the remaining formula corresponds to a learnt clause falsified at the current node.

Observation 3. *Let π be a partial assignment to the variables of $\mathcal{F}$ and let κ be a core of $\mathcal{F}|_\pi$. If $\kappa' \subseteq soft(\mathcal{F})$ is the set of clauses in $\mathcal{F}$ corresponding to the clauses in κ, then there is a resolution derivation of a clause c from $\kappa' \cup hard(\mathcal{F})$ such that π falsifies c.*

Proof. The steps performed in a refutation of the core κ can be copied starting with the clauses in κ' in order to derive the desired learnt clause c. □

Thus, given a core of the remaining formula and its refutation, we can save the set of original clauses $\kappa' \subseteq soft(\mathcal{F})$ and a learnt clause c that can be derived from them. If c is falsified by the current assignment at any node of the future search, it implies that at least one of the clauses in κ' will also be falsified at every leaf node in the subtree. So the falsification of the learnt clause c indicates when the contextual core κ' can be used, and all contextual cores that are relevant at a node can be combined in a MCHS computation to derive a lower bound.

Indeed, this method is complete in the sense that if we had access to *all* possible learnt

clauses that are falsified at node n, then a MCHS computation would allow $\mathit{mincost}(n)$ itself to be computed.

Definition 21 ($\mathcal{P}_{all}$, ic)**.** *Let $\mathcal{F}$ be a* maxsat *instance and let n be a node of the Branch and Bound search. Then $\mathcal{P}_{all}(\mathcal{F}, n)$ is the set of all resolution proofs from $\mathcal{F}$ (ignoring clause weights) that derive a clause falsified by the assignments made at node n. Given a proof $\phi \in \mathcal{P}_{all}(\mathcal{F}, n)$, let $ic(\phi)$ be the set of clauses of $\mathit{soft}(\mathcal{F})$ used in ϕ.*

Proposition 15. *Let $\mathcal{K} = \{ic(\phi) : \phi \in \mathcal{P}_{all}(\mathcal{F}, n)\}$ and let hs be a MCHS of $\mathcal{K}$. Then $\mathit{mincost}(n) = \mathit{cost}(hs)$.*

Proof. The proof of this proposition is more complicated than the proof of Proposition 2 on page 39. By Proposition 14, $\mathit{mincost}(\mathcal{F}|_n) = \mathit{mincost}(n)$. Proposition 2 then shows that if we have all cores of $\mathcal{F}|_n$, and a MCHS H for them, $\mathit{cost}(H) = \mathit{mincost}(\mathcal{F}|_n) = \mathit{mincost}(n)$. Now all that remains to be done is to prove that $\mathit{cost}(H) = \mathit{cost}(hs)$.

First, Observation 3 shows that all cores of $\mathcal{F}|_n$ can be converted to equivalent proofs in $\mathcal{P}_{all}(\mathcal{F}, n)$ by adding back all of the falsified literals to the clauses of the refutation. Thus a MCHS for $ic(\mathcal{P}_{all}(\mathcal{F}, n))$ is at least as big as a MCHS of all cores of $\mathcal{F}|_n$, i.e. $\mathit{cost}(hs) \geq \mathit{cost}(H)$.

The other direction is more complex, but the same argument as Proposition 2 can be used to show that if $\mathit{cost}(hs) > \mathit{cost}(H)$ then there must be a hitting set H' of the cores of $\mathcal{F}|_n$ which when converted to a set of clauses of $\mathcal{F}$ (by adding back the falsified literals) is not a hitting set of $ic(\mathcal{P}_{all}(\mathcal{F}, n))$.

Hence, there is a proof $\phi \in \mathcal{P}_{all}(\mathcal{F}, n)$ not covered by H'. With a more complex transformation, ϕ can then be converted into a refutation of $\mathcal{F}|_n$ by removing all satisfied clauses and falsified literals, and then fixing all of the now broken resolution steps. (For example, the literal being resolved on might have been removed from one of the clauses, or one of the clauses might have been satisfied). The conversion of ϕ is a refutation of $\mathcal{F}|_n$ not hit by H', contradicting that H' exists. □

We are unlikely to have access to all of the proofs in $\mathcal{P}_{all}(\mathcal{F}, n)$, but any subset can be used to produce a lower bound on $mincost(n)$.

Corollary 2. *Let $\mathcal{K} \subseteq \{ic(\phi) : \phi \in \mathcal{P}_{all}(\mathcal{F}, n)\}$ and let hs be a MCHS of $\mathcal{K}$. Then $mincost(n) \leq cost(hs)$.*

In summary, at any node n of the search tree some set of clauses will be falsified. These could be either original clauses of $\mathcal{F}$ or learnt clauses. By keeping track of the soft input clauses used to derive each falsified learnt clause a hitting set problem can be set up. By Corollary 2 the minimum cost hitting set provides a lower bound on the minimum cost assignment that can be achieved below node n, $mincost(n)$.

Our lower bound approach requires some additional information to be stored along with the learnt clauses. Specifically, for each learnt clause the set of original soft clauses or previously learnt clauses used in its derivation must be stored with it. However, the overhead of saving and accessing this information is very low. We do not have to remember the entire resolution proof of the learnt clause, just a list of pointers to the original clauses of $\mathcal{F}$ and to other previously learnt clauses from which it was derived. Furthermore, a watched literals scheme over the learnt clauses can be used so that the additional information stored with a learnt clause is only accessed when the the learnt becomes falsified.

Two problems remain. First it may be time-consuming to compute a minimum cost hitting set as this is an NP-hard problem in itself. The next section presents some ways of computing lower bounds on the minimum cost hitting set, which in turn act as lower bounds on $mincost(n)$. Second, in Section 6.5 we develop techniques to generate learnt clauses either in a preprocessing step, or dynamically during the search.

6.4 Lower Bounding the Minimum Cost Hitting Set

We use the two subsumption rules defined in Section 4.1.1 to simplify the hitting set problems. These two rules define a propagation scheme, since one application of a rule can enable additional applications. Simplification continues until neither of these rules can be applied again. It is also possible to remove from the hitting set problem any clauses that are satisfied at the current node, and we do this first since it may enable some more subsumptions. These simplifications often generate a collection of disjoint hitting set problems. If so, we solve or approximate the disjoint problems independently and then add the results.

6.4.1 Heuristic Lower Bounds

Two heuristics for lower bounding the minimum cost hitting set are considered.

```
H1(K)
1. LB = 0
2. while K ≠ ∅
3.    choose κ ∈ K
4.    LB += min_{c∈κ} wt(c)
5.    S = {κ' ∈ K | κ ∩ κ' ≠ ∅}
6.    K = K − S
7. return LB
```

This heuristic first chooses some set to hit, κ, and adds the cost of its minimal cost element to the lower bound. Then it removes κ and all other $\kappa' \in \mathcal{K}$ that share an element of κ. It repeats this loop until there are no more sets to hit.

The intuition behind this heuristic is simple: κ can be hit by selecting a min-cost element. But any other element from κ could have been chosen instead to hit κ. Hence, the most that could have been hit is all other sets that intersect κ. The heuristic conservatively estimates that indeed all of these sets were hit with κ's min-cost element. Note

that the heuristic can yield different values depending on which κ is chosen.

We use two different selection policies to choose the next κ on line 3. For MAXSAT with non-uniform weights on the soft clauses, κ is chosen to be the set whose min-cost element is of maximal cost. If, on the other hand, all the soft clauses have weight 1, κ is chosen to be a set that intersects with the fewest other sets (i.e. whose set S in line 5 is of minimal cardinality). However, other natural policies do exist.

This heuristic inherently takes advantage of any disjoint subproblems. In particular, if the hitting set problem has been broken up into k disjoint subproblems, `H1` will return a bound that is no worse than the sum of a minimal cost element from each subproblem.

For the second heuristic, for an element c let $nbrs(c) = \{\kappa : \kappa \in \mathcal{K} \wedge c \in \kappa\}$ be the sets hit by c and let $deg(c) = |nbrs(c)|$.

```
H2(K)
1. LB = 0
2. n = |K|
3. while n > 0
4.     c = c ∈ K that minimizes wt(c)/deg(c)
5.     if deg(c) < n OR uniform weights
6.       LB += wt(c)
7.     else LB += n × wt(c)/deg(c)
8.     n -= deg(c)
9.     remove c from κ for all κ ∈ K
10. return LB
```

This heuristic generalizes one given in (Petit et al., 2003). It operates by selecting an element with lowest weight over degree. These elements hit the most sets on a minimal cost per set basis. Sets are chosen in this way until the sum of their degrees equals or exceeds the total number of sets to hit. However, in the case of weighted elements, only part of the weight of the last element selected can be counted (line 7).

Proposition 16. *(Davies et al., 2010) Both `H1` and `H2` return a lower bound on the cost*

of the minimum cost hitting set.

These two heuristics are incomparable. That is, on some problems `H1` provides a better bound than `H2` and vice versa on other problems.

For example, let $\mathcal{K} = \{\kappa_1, \kappa_2, \kappa_3\}$, where $\kappa_1 = \{c_1, c_2\}$, $\kappa_2 = \{c_1, c_3\}$ and $\kappa_3 = \{c_2, c_3\}$ and all elements have equal weight of 1. No matter which set $\kappa \in \mathcal{K}$ is first chosen by `H1`, it will only perform a single iteration and produce a lower bound of 1. However, `H2` can pick any two elements for a lower bound of 2.

On the other hand, let $\mathcal{K} = \{\kappa_1, ..., \kappa_6\}$ be a hitting set problem over a universe of three equally weighted elements, where $\kappa_1 = \kappa_2 = \{c_1\}$, $\kappa_3 = \{c_1, c_2\}$, $\kappa_4 = \{c_2\}$, $\kappa_5 = \{c_2, c_3\}$ and $\kappa_6 = \{c_3\}$. Then $nbrs(c_1) = \{\kappa_1, \kappa_2, \kappa_3\}$, $nbrs(c_2) = \{\kappa_3, \kappa_4, \kappa_5\}$, and $nbrs(c_3) = \{\kappa_5, \kappa_6\}$. In this case, `H1` can pick $\{\kappa_1, \kappa_4, \kappa_6\}$ for a lower bound of 3. However `H2` is forced to pick $\{c_1, c_2\}$ for a lower bound of 2.

Unfortunately both heuristics can yield arbitrarily bad approximations.

Theorem 6. *(Davies et al., 2010) For a MCHS instance $\mathcal{K}$ let the cost of its solution be denoted mincost($\mathcal{K}$), and let `H1`($\mathcal{K}$) and `H2`($\mathcal{K}$) be the lower bounds computed by the two heuristics. Then for any $\epsilon > 0$, there exists $\mathcal{K}$, $\mathcal{K}'$ such that `H1`($\mathcal{K}$)/mincost($\mathcal{K}$) $\leq \epsilon$ and `H2`($\mathcal{K}'$)/mincost($\mathcal{K}'$) $\leq \epsilon$.*

6.4.2 Linear Relaxation Lower Bound

The quality of the lower bound on the minimum hitting set dictates how soon the search can backtrack from a non-optimal partial assignment. Furthermore, by recomputing the lower bound at each node during backtrack,[2] it also impacts how far the search can backtrack (as long as $LB \geq UB$). In light of this and Theorem 6, it may be desirable to use more powerful techniques to compute the exact value of the minimum hitting set at strategic points during search.

[2]Note that it may not be sufficient to simply reuse the lower bound that was computed the first time the node was reached, as additional learnt clauses may have been added to the hitting set instance since then.

One way to solve the MCHS problem to optimality is to encode it as an integer program, and solve it using a MIP solver such as CPLEX. Or, we can use CPLEX to solve the linear relaxation instead, since it is much cheaper to compute and also gives a valid lower bound on the minimal cost hitting set.

To balance the trade off between the quality of the bound and the computational cost required, the following strategy is adopted: first heuristics `H1` and `H2` are computed and their maximum used as initial lower bound LB. If $LB < UB$ but $LB/UB \geq \alpha$, for some tuned parameter α, then the linear relaxation is solved. Finally, if this is still insufficient to exceed UB and if the size of the hitting set problem is less than some other tuned parameter β, the integer program is solved.

6.5 Learning Clauses

It remains to consider how to generate the learnt clauses that will be used in the hitting set lower bound. The first method we investigated is to perform a preprocessing step to learn clauses using relaxed DPLL search (Kroc, Sabharwal, and Selman, 2009). However, we soon discarded this limited approach in favour of turning our attention to techniques that can learn clauses during the Branch and Bound search itself.

6.5.1 Relaxed DPLL Preprocessing

In (Kroc et al., 2009), a relaxed DPLL search is used to produce a good upper bound on the cost of a solution to an unweighted MAXSAT instance. Kroc et al. modify a SAT solver to ignore the first k conflicts along a branch, and only learn and backtrack from the next conflict if one occurs. Such a procedure can be used as a preprocessing step to generate learnt clauses. It is possible to extend their idea to the weighted case (and hard clauses) by treating k as a bound on the total cost to ignore. Since the relaxed search

performs unit propagation over all clauses, both soft and hard, the clauses learned may be derived from soft clauses of $\mathcal{F}$. The set of soft clauses of $\mathcal{F}$ used to derive each learnt can be obtained by modifying MINISAT's clause learning procedure (Gelder, 2009).

The number of learnt clauses produced by this process can vary significantly. In some cases, the MAXSAT instance is too easy to refute, and very few or no learnt clauses will be produced. For example, if the MAXSAT instance contains soft unit clauses, the relaxed DPLL search will immediately propagate them at the root. It is not unusual that propagating the soft unit clauses of $\mathcal{F}$ falsifies a hard clause. In this situation the relaxed DPLL search will terminate at the root, since infinite cost has been incurred (exceeding any threshold k) and no backtracking is possible. In this case, we could consider modifying the relaxed DPLL search to ignore the soft unit clauses of $\mathcal{F}$. However, this modification would require additional techniques to prevent search from re-visiting the same assignments, and was not investigated further for this reason.

In other cases, the number of learnt clauses generated by such preprocessing can be too large. The number of learnt clauses can be limited simply by terminating the process at any point, or by using the SAT solver's default criteria to prune the learnt clause database whenever it grows too big. However, we propose a different technique to prune the learnt clause database, that is designed with our purpose in mind. Each learnt clause is assigned a weight equal to the weight of the minimal weight soft clause of $\mathcal{F}$ used to derive it. The score of a learnt clause is then equal to its normalized weight (the clause weight divided by an upper bound on the cost of the MAXSAT solution) minus its normalized length (the clause length divided by the number of variables in the MAXSAT instance). The learnt clauses with highest scores, over all clauses learnt, are chosen to be output. This selects the clauses that are short, and therefore have a chance to contribute to the lower bound higher up in the search tree. Clauses derived from high weight clauses are also favoured, in the hope that they will contribute significantly to the hitting set lower bound.

We implemented these techniques on top of MINISAT, to create a preprocessor that given a MAXSAT instance outputs a collection of learnt clauses and the sets of soft clauses used to derive them. We implemented a preliminary version of Branch and Bound search for MAXSAT that reads in a collection of such learnt clauses and uses them to calculate our hitting set bounds. However, our experiments suggested that the preprocessing approach would not lead to a robust MAXSAT solver. Certainly, this approach does not allow the main Branch and Bound search to learn from its own progress to better prune its future search space.

6.5.2 Learning Clauses During Branch and Bound

The approach we take to learn clauses during Branch and Bound is based on performing *trial* unit propagations at each node of search. These trial unit propagations (TUP) are applied after all hard inferences have been performed. They allow soft unit clauses to be propagated temporarily, in order to derive conflicts.

At every node n of our Branch and Bound search, unit propagation is applied over the hard clauses.[3] If a conflict is found, a new hard clause is learnt and backtracking is performed like in a regular SAT solver (see Appendix B). These techniques are well-known and used by other state-of-the-art Branch and Bound solvers for MAXSAT (see Section 2.5.1).

If hard inference does not cause the search to backtrack from n, then a phase of *trial unit propagation* (TUP) begins. The TUP stack is initialized with all soft clauses that are unit or falsified in $F|_n$, as is done in MINIMAXSAT (Heras et al., 2008). TUP involves unit propagating soft clauses as well as hard, so any literals set during the TUP phase must be undone before continuing search with another decision. However, if a clause is falsified during TUP, this conflict can be analyzed using standard techniques to produce a learnt clause with the desirable property of being falsified at n, *before* TUP.

[3] As the upper bound is refined, more clauses become hard.

Thus it may contribute to increasing the hitting set lower bound at the current node. Therefore, the learnt clauses produced are immediately relevant to the search, in contrast to those obtained from a preprocessing step. The learnt clause c, together with the set of soft clauses of $\mathcal{F}$ used to derive it, is saved upon backtrack and can be used in future search whenever c is falsified, to contribute to the hitting set lower bound. The learnt clauses also participate in future TUP phases, so that clauses can be learned from others, contributing to the power of these clauses to prune the search.

TUP continues to propagate and learn clauses until no more new falsified clauses are found. Thus many learnt clauses can be produced at each node of the search. The lower bound is updated once after TUP is finished. In future work we intend to investigate ways to limit or guide the TUP phase, in order to find a better trade-off between the strength of lower bound produced and the time spent in TUP.

Turning Off Clauses

TUP learning, if implemented as described above, can produce duplicate learnt clauses.[4] In order to prevent duplicates from creating overhead, they are prevented from being learned in the first place. This is achieved by "turning off" for TUP, one of the clauses used to derive each existing falsified learnt clause c, for the duration of time c remains falsified. A turned-off clause can't be used to derive any more learnt clauses, until it is turned back on. Of course, this policy may reject new learnt clauses that aren't actually duplicates of an existing one. This is undesirable since it may reduce the strength of the lower bound. To limit the negative impact on heuristics `H1` and `H2`, we always choose to turn off the clause of least weight or greatest degree. In future work we intend to try a less cautious method of avoiding duplicate learnt clauses: we can turn off clauses only at decision levels greater than k, and rely on a general scheme of learnt clause database

[4] A learnt clause is only considered a duplicate of another if they were derived from the same set of soft clauses of $\mathcal{F}$, since otherwise each can contribute to the lower bound under different circumstances depending on the structure of the hitting set problem.

reduction to limit the total number of learnt clauses.

Similarly, when a soft clause c is falsified by the current prefix, all of the learnt clauses it has derived are turned off. This prevents clauses being learned that can't currently contribute to the lower bound. To see this, note that any learnt clause c' whose derivation is blocked in this way, would also have been derived indirectly from c. While c is falsified, its cost is already counted and does not contribute to the hitting set lower bounds. Therefore, adding c' to the hitting set problem can not increase the lower bound while c is false.

6.6 Related Work

If no clauses are turned off for TUP, and the optimal MCHS is calculated for the lower bound, our lower bound technique subsumes all existing unit propagation based bounds in the MAXSAT literature (Li et al., 2007; Heras et al., 2008; Xing and Zhang, 2005). If clauses are turned off for TUP, the resulting lower bound will be at least as good as the disjoint inconsistent subformulas bound (Li et al., 2007), since turning off clauses in one inconsistent subformula does not prevent disjoint subformulas from being refuted.

A related lower bound was used in a Branch and Bound solver for MaxCSP (Petit et al., 2003). Their bound differs from our approach in several ways, such as how the conflicts are detected and how the MCHS problems are rendered tractable. Their solver also does not perform learning, as all of the conflicts are discarded upon backtrack.

6.7 Experimental Results

We modified the SAT solver MINISAT (Eén and Sörensson, 2003) to perform a Branch and Bound search for MAXSAT. We implemented the variable ordering heuristics used by MINIMAXSAT, and the clause learning procedure was strengthened with Failed Literal Detection (Heras et al., 2008). The Dominating Unit Clause rule, which allows a literal

to be instantiated if the weight of its unit clauses is at least the weight of all clauses containing its negation, is used to simplify the theory at each node (Zhang et al., 2003). The resulting solver is called minimaxhs. In general, we found that minimaxhs can not compete with state-of-the-art Branch and Bound maxsat solvers. Although our solver is sometimes able to compute better lower bounds that result in smaller search trees, the overhead of our techniques results in poor overall performance. However, we believe that the implementation can be improved, e.g., by limiting the number of learnt clauses generated at each node.

Since the overall performance of minimaxhs is not competitive with state-of-the-art solvers, we report only preliminary experiments that show the potential of the minimaxhs solver.

We collected a set of 378 instances from the maxsat Evaluations (Argelich et al., 2007–2012) by identifying benchmark families in which minimaxsat is unable to solve some instances, and where our solver minimaxhs can outperform minimaxsat on at least some problems. These benchmark families were chosen to illustrate the cases in which our method can display interesting behaviour. The selected instances represent all weight types (maxsat, partial maxsat etc.) and categories (Random, Crafted, Industrial). The experiments were conducted on a dual-core 2GHz AMD Opteron processor with 3GB of RAM, and all experiments were run with a 1200 second timeout. These experiments were performed for the paper (Davies et al., 2010).

6.7.1 Comparing the Lower Bound Heuristics

The first set of experiments investigates the performance of the two hitting set heuristics, `H1` and `H2`, in the context of providing a lower bound during Branch and Bound search. We ran the search by computing both heuristics at every node of the search, and taking their maximum as the lower bound. The number of times each of the two heuristics provided different bounds was counted, and the relative amount by which the winner

was better was calculated. We discarded easy instances that were solved by all methods in under 100 decisions, leaving 226 instances. The results are summarized in Table 6.1. The first column specifies the heuristic that was used, either `H1` or `H2` alone, or their maximum. The second column shows the percentage of all lower bounds calculated for which the one heuristic gave a larger bound than the other (averaged over all instances). Whenever one heuristic gave a strictly larger bound than the other, the relative difference (e.g. `H1`/`H2` if `H1` was the larger) was measured; the third column reports this averaged over all instances. The average number of decisions and average runtime are shown in columns four and five (these averages are taken over the subset of the 226 instances that all three lower bound methods could solve, which included 112 instances). The total number of instances solved (out of 226) using each method is included in the last column. These results show that it is best to calculate both lower bounds and take their maximum, since they are both cheap to calculate and can solve more problems when combined.

LB Heuristic	Freq	Size	Decisions	Time (s)	Num Solved
`H1`	50	1.15	36280	49	115
`H2`	6	1.09	40115	50	115
max(`H1`,`H2`)	–	–	36192	49	117

Figure 6.1: Comparison of the `H1` and `H2` lower bounds during search, on the 226 instances that required more than 100 decisions to solve. The 'Freq' column refers to the percentage of all lower bounds calculated for which the heuristic gave the larger bound, averaged over all instances. The 'Size' column gives the average factor by which the bound was larger. The Decisions and Time (s) columns report the average number of decisions and the average runtime, on the set of 112 instances that all three lower bound techniques solved. The last column specifies the number of instances solved out of the 226.

Next, we present results of experiments to investigate the trade-off between using the `H1` heuristic, and solving the linear program for the hitting set problem using CPLEX. The dynamic addition and removal of variables and constraints from the CPLEX model can limit the efficiency of this approach, and the results confirm that the added strength of the LP lower bound comes with the price of spending more time at each node. We ran

MINIMAXHS with the H1 lower bound alone, and then with only the LP lower bound. The results are shown in Table 6.2 for the 97 instances that were solved by both methods. By using the LP lower bound, on average 28% fewer decisions were made by MINIMAXHS. The reduction in decisions did pay off on 37 instances, for which the total runtime was reduced by using the LP bound. However in the majority of cases, the LP bound increases the runtime by an average of about 30%. In general, the extra computational cost does not pay off, since using the stronger LP bound solves 10 fewer problems. These results demonstrate that a hybrid approach, using the stronger bounds at judicious points during search to exceed the UB, is a well-justified direction for future work.

LB Heuristic	Decisions	Time (s)	Num Solved
CPLEX LP	25296	48	105
H1	35059	15	115

Figure 6.2: Comparison of the H1 and LP lower bounds during search. The average number of decisions and runtime (over instances both methods solved), and the number of instances solved is shown.

6.7.2 Solving Without Search

Finally, we highlight an interesting observation about the behaviour of MINIMAXHS on some instances.

MINIMAXHS applies an initial phase of Failed Literal Detection (FLD) at the root of the Branch and Bound search to find a collection of learnt clauses. We combine FLD with TUP in the following procedure. Each literal in $\mathcal{F}$ is assigned in turn, and the formula is reduced using unit propagation to identify hard inferences, followed by TUP. If a clause is falsified, a unit or empty clause is learned, before undoing all propagations and moving on to probe the next literal. On some instances, the set of clauses learnt during probing is informative enough that the lower bound computed by finding their MCHS will equal an upper bound provided by one run of ubcsat (Tompkins and Hoos, 2004), that is, these instances are solved without search by MINIMAXHS. This occurs on

at least 16 problems, presented in Table 6.3. Note that we only report instances where the Branch and Bound solver MINIMAXSAT required some branching to solve the problem. Here, CPLEX is able to solve the initial MCHS problem exactly, because it is sufficiently small. These results are not surprising given the success of MAXHS that we've seen in previous chapters. We also observe that the initial processing performed by MINIMAXHS (FLD and a MCHS computation) can be quite expensive, even compared to exploring a moderately large search tree with MINIMAXSAT. However, the potential for saving a significant amount of search via our methods does exist, as illustrated by, e.g., the results on instance norm-mps-v2-20-10-stein27.

Year	Type	Name	Optimum	MINIMAXSAT Decisions	Our Decisions	MINIMAXSAT Time (s)	Our Time (s)
2007	wpms	8.wcsp.log	2	1	0	0.05	0.05
2007	wpms	norm-mps-v2-20-10-stein15	9	2191970	0	3.8	0.07
2007	wpms	norm-mps-v2-20-10-stein27	18	–	0	>1200	0.59
2007	wpms	norm-mps-v2-20-10-stein9	5	230	0	0.12	0.14
2008	pms	norm-fir01_area_delay	5	14	0	0.14	0.12
2008	pms	norm-fir02_area_partials	19	38	0	0.16	0.06
2008	pms	norm-fir04_area_partials	30	13	0	0.12	0.6
2008	wms	frb10-6-1	50	755	0	0.27	0.23
2008	wms	frb10-6-2	50	678	0	0.13	0.26
2008	wms	frb10-6-3	50	1302	0	0.13	0.23
2008	wms	frb10-6-4	50	580	0	0.29	0.3
2008	wms	frb15-9-1	120	387470	0	1.37	3.76
2008	wms	frb15-9-2	120	206845	0	2.29	4.6
2008	wms	frb15-9-4	120	199365	0	2.24	5.77
2008	wms	frb15-9-5	120	271024	0	1.27	5.59
2009	wpms	warehouse0.wcsp	328	46	0	0.11	0.12

Figure 6.3: Comparison of MINIMAXSAT and MINIMAXHS on instances that MINIMAXHS can solve without branching.

6.8 Conclusion

This chapter introduced an innovative approach for MAXSAT solving, with potential for practical impact based on generating bounds from unrestricted clause learning for MAXSAT. Although it may always be necessary to use a restricted version on real problems, it was argued that this framework provides new insight into how strong lower bounds can be made practical, for example, by being smart about which soft clauses are learnt, or by approximating the minimum hitting set well. In addition to these contri-

butions, two heuristics for the weighted hitting set problem were presented, and shown to be effective in a novel context. Based on a preliminary implementation, it is clear that the primary challenge in bringing this technique to the state-of-the-art in practical performance, will be to develop methods to learn the best clauses to prune the search tree.

Chapter 7

Conclusion

This thesis presented practical algorithms to solve optimization problems expressed as MAXSAT. MAXSAT uses propositional clauses with weights to express the constraints of the optimization problem and the costs associated with violating these constraints. Previously, many algorithms to solve MAXSAT were proposed and implemented in MAXSAT solvers. Many of these solvers were successfully applied to solve a variety of important optimization problems in computer science and industry. These existing MAXSAT solvers follow one of two paradigms: Branch and Bound search, or solving a sequence of decision problems. Most of the existing MAXSAT solvers borrow techniques from SAT solving, either to calculate bounds in Branch and Bound, or to solve the decision problems by translating them to SAT and applying a SAT solver as a black-box.

However, in real-world applications, optimization problems are usually modelled and solved as MIP, not MAXSAT. MIP uses arbitrary linear constraints and a linear objective function to model optimization problems. In the introduction we pointed out that MIP can be used to model and solve MAXSAT instances as well, and in Chapter 3 we found that a MIP solver like CPLEX can sometimes outperform dedicated MAXSAT solvers. Yet this approach is not able to solve Industrial MAXSAT instances, e.g. those arising from Electronic Design Automation, as efficiently as MAXSAT solvers that exploit a SAT solver

to solve a sequence of decision problems, as we observed from Table 3.3.

Since MAXSAT combines aspects of both SAT and optimization, and mature technologies exist for each of these domains, in this thesis we investigated new methods to exploit both SAT solvers and MIP solvers as black-boxes in order to solve MAXSAT robustly and efficiently. The proposed MAXHS approach automatically and incrementally splits the MAXSAT problem into two parts, given to the SAT and MIP solvers respectively, and facilitates communication between the two sub-solvers so that they work together to solve the MAXSAT problem. We demonstrate through an extensive empirical evaluation that our new MAXSAT solver based on the MAXHS approach is more robust than any existing MAXSAT solver. Furthermore, the hybrid MAXHS solver performs significantly better than using a MIP solver alone to solve MAXSAT. Thus, we have achieved the desired goal of combining the strengths of both SAT and MIP algorithms to better solve MAXSAT.

The thesis defined the basic MAXHS algorithm in Chapter 3, where it was then established that the behaviour of MAXHS depends on three factors: the time taken by the SAT solver, the time required by the MIP solver, and the number of iterations (i.e., calls to the SAT and MIP solvers). We observed that in practise, the time taken by the SAT solver is much less than the time taken by the MIP solver. So we investigated ways of using the SAT solver to improve the information provided to the MIP model, and showed empirically that this usually results in a decrease in the total runtime since each call to the MIP solver is more effective.

In Chapter 4 we investigated several techniques to enrich the MIP model built by MAXHS, motivated by two observations. The first observation was from the experimental results of Chapter 3, which showed that the MIP solver CPLEX can solve many MAXSAT instances better than MAXHS or any other MAXSAT solver. The second observation was that the MAXSAT problem imposes many additional constraints on the MIP sub-problem. Therefore, we proposed three techniques to enrich the information in the MIP model. First, we identified a so-called *realizability* condition on the solutions to the MIP model.

Second, we developed a method that allows the SAT solver to discover more general *non-core* constraints to give to the MIP model. And third, we proposed to *seed* the MIP model with a collection of constraints derived from the original MAXSAT formula. The last two ideas lead to a significant improvement in the overall performance of MAXHS, and the resulting version of our solver is robust across Crafted and Industrial instances.

Chapter 5 considered an orthogonal method of increasing the information given to the MIP model. We showed how MAXHS can sometimes use approximate solutions to the MIP sub-problem in order to reduce the number of expensive optimizations. The approximate solutions are used to drive the SAT solver's discovery of more constraints, which allows a large number of constraints to be generated very quickly. We found empirically that these constraints greatly improve the effectiveness of the optimal MIP solving episodes, and this trade-off often results in a significant reduction in total runtime. We also showed that combining the use of non-optimal hitting sets with the seeding technique of the previous chapter leads to even greater improvements. The resulting version of our MAXHS solver solves more problems overall than any other tested approach, and is robust for both Crafted and Industrial categories.

Finally, in Chapter 6 we proposed to use some of the MAXHS ideas in a Branch and Bound solver for MAXSAT. Existing Branch and Bound solvers for MAXSAT suffer because the most powerful technique from SAT solving, clause learning, is not sound for MAXSAT. The techniques we introduced in this chapter are therefore important as they allow the search to learn to improve its bounds as it progresses. We showed that the new lower bound methods can greatly reduce the size of the search tree, but further experimentation is needed to explore the trade-off between the time spent to calculate the bounds and their strength.

This thesis introduced the MAXHS approach, which is a hybrid framework that offers much flexibility to design different algorithms for MAXSAT. The thesis proposed and evaluated several ways to enhance the basic MAXHS algorithm, and showed that it is pos-

sible to develop a very effective, state-of-the-art MAXSAT solver based on this framework. However, the MAXHS approach defines a very rich design space, much of which is yet to be explored. Indeed, there are many promising directions for future work.

It is likely that the robustness of the MAXHS approach can be greatly improved by tailoring the various techniques introduced by this thesis on an instance-specific basis, perhaps by employing an Instance Specific Algorithm Configurator like Hydra (Xu, Hoos, and Leyton-Brown, 2010). The MIP solver offers a richer modelling language than MAXSAT that could be further exploited to represent non-logical constraints, so methods to identify such constraints in a MAXSAT instance could be developed.

In addition, it will be interesting to discover if there are new applications that can be solved via the MAXHS approach. By combining MIP and SAT models, MAXHS is well suited to applications where the original optimization problem contains different aspects that are naturally expressed as linear and logical constraints. Although optimization problems with general constraints can be solved as MIP or Weighted Constraint Satisfaction (WCSP), we believe that the power of MIP and SAT solvers, when effectively combined by MAXHS, provides a uniquely powerful approach to solve real-world problems.

Appendices

Appendix A

Basic Terminology

A propositional variable v can take on two values: *true* and *false*. A *literal* is a propositional variable v or its negation $\neg v$ ($\neg\neg v = v$). A *clause* is a set of literals and a formula in Conjunctive Normal Form (*CNF*) is a set of clauses. Unless otherwise noted, n and m refer to the number of variables and clauses in a formula respectively.

If π is a set of literals such that no variable appears more than once in π, then π is a *truth assignment*. If π is a truth assignment that mentions every variable in a formula, then π is a *complete* truth assignment wrt that formula, and otherwise it is a *partial* assignment. A clause C is satisfied by a truth assignment π if $\pi \cap C \neq \emptyset$. A clause is *falsified* by a truth assignment π if $\neg\ell \in \pi$ for all $\ell \in C$. A CNF formula Φ is satisfiable if there exists a complete truth assignment that satisfies all of its clauses. If no such truth assignment exists, Φ is unsatisfiable. An unsatisfiable subset of the clauses of Φ is called an *UNSAT core*. The SAT problem is to determine whether a given CNF formula is satisfiable or not. This decision problem is NP-complete (Cook, 1971).

If v and $\neg v$ appear in the same clause, then the clause is a *tautology* (i.e. it is satisfied by every truth assignment). If a clause contains no literals, it is called an *empty clause* and it is falsified by every truth assignment. A *unit* is a clause containing only one literal, and a *binary clause* is one that contains two literals. A clause C_1 *subsumes* another clause

C_2 if $C_1 \subseteq C_2$.

The *clause:variable ratio* (the number of clauses divided by the number of variables) is an important property of a CNF formula. Formulas with higher clause:variable ratios are considered to be more *constrained*, and are in general more likely to be unsatisfiable.

Given a CNF formula Φ and a literal ℓ, $\Phi|_\ell$ is the formula that results from removing every clause containing ℓ and removing $\neg\ell$ from every clause where it appears. This transformation is called *reducing* the formula by ℓ, or *instantiating* ℓ.

Starting with a CNF formula Φ, *Unit Propagation* (UP) is the process of repeatedly selecting a unit clause and instantiating its literal until no more unit clauses are left in the formula or an empty clause is generated. If UP produces an empty clause, then it said that UP has found a *conflict*. This implies that the original formula Φ is unsatisfiable, because UP is sound for SAT.

Resolution is an inference rule that preserves the satisfiability of a CNF theory, and is therefore sound for SAT. The resolution rule can be applied to two clauses C_1 and C_2 in a CNF Φ, if the clauses are *clashing*, i.e. there exists a literal $\ell \in C_1$ such that $\neg\ell \in C_2$. In this case, resolution infers the resolvant clause $C_1 \cup C_2 \setminus \{\ell, \neg\ell\}$ which is added to Φ. A *resolution derivation* of a clause C from a formula Φ is a sequence of clauses $C_1, ..., C_{k-1}, C_k = C$ where each C_i is either a clause in Φ or is the resolvant of two clauses appearing previously in the sequence. A resolution *refutation* is a resolution derivation of the empty clause. Since resolution preserves satisfiability, if Φ has a refutation it implies that Φ is unsatisfiable. Resolution is a complete inference rule for SAT, meaning that there is a resolution refutation for every unsatisfiable CNF formula.

Appendix B

DPLL SAT Solvers

The most successful SAT solvers are based on the *DPLL* algorithm, shown in Algorithm 11. The algorithm begins with the empty partial assignment, and extends it by alternately making a *decision* (i.e. assigning some chosen literal), and performing unit propagation, until a clause is falsified or a solution is obtained. When the current partial assignment falsifies a clause, the search backtracks, which involves undoing one or more of the most recent decisions before continuing the search. At this point, a new *learnt* clause, entailed by the original theory, can be derived that may help to prune the future search.

The algorithm uses a *watched literals* data structure, which for every literal ℓ has a list of clauses whose first or second literal is ℓ. During the search, the invariant that the first two literals in every clause are unassigned or satisfied must be maintained when possible. The purpose of the watched literals is to allow clauses that are made unit or falsified by the current assignment to be identified efficiently.

DPLL begins on lines 3-7 of Algorithm 11 by initializing the watch lists, and assigning the literals in any unit clauses of the input theory. The assign function takes a literal ℓ and a reason for its assignment (NULL or a clause that has become unit on ℓ) as arguments. Assigning a literal involves recording its variable's truth value, as well as the

decision level at which it was assigned, and its reason. These three pieces of information, for each variable, are stored in global arrays *value*, *level* and *reason*, which are assumed to be available to all subfunctions. The assign function is also responsible for pushing the literal onto a global stack, UPstack, used to implement unit propagation.

At every step, the algorithm performs unit propagation (line 9), according to Algorithm 12. If UP returns a falsified clause C, it means UP has discovered a conflict and the current partial assignment can not be extended to a solution to Φ. In this situation, *clause learning* is applied on line 13, to derive a new learnt clause L from the conflict. The learnt clause is falsified by the current assignment, and entailed by the input theory Φ. Many such clauses could be generated, but a popular choice is the *1-UIP* clause, which can be derived using the procedure in Algorithm 13.

After learning a new clause and installing it on the watch lists, the search backtracks to the decision level at which the learnt clause is unit, assigns its remaining literal, and continues the search (lines 15-16). This style of backtracking is called non-chronological backtracking since more decisions than the last one may be deleted. The backtrack(d) function on line 15 just resets the values of the literals whose decision levels are greater than the parameter d and updates dlevel; backtracking is a very cheap procedure.

If no conflict is discovered by UP at this step, and the current assignment is complete, it means that the current assignment is a solution to Φ and the search is finished (lines 18-19). Otherwise, the algorithm extends the current partial assignment, by choosing and assigning another decision literal (lines 20-22). This marks the start of a new decision level, so the decision level is incremented on line 21.

The DPLL algorithm actually creates a resolution refutation if the input formula is UNSAT. Intuitively, if the algorithm returns UNSAT on line 12, it is because an empty clause has been derived through a number of calls to Algorithm 13. Each call to Algorithm 13 is clearly a resolution derivation of a new learnt clause from existing learnt clauses and input clauses. If all these sub-derivations are saved, they can be put together

to reconstruct the derivation of the empty clause. So DPLL can be seen as a proof system for SAT, that can produce refutations that are as compact as is possible through Resolution (Hertel, Bacchus, Pitassi, and Gelder, 2008; Pipatsrisawat and Darwiche, 2009).

Algorithm 11: The DPLL algorithm for solving SAT.

```
1  SAT-DPLL (Φ)
   /* Initialize the current decision level and unit propagation stack */
2  dlevel = 0, UPstack = ∅
   /* Assign the unit clauses of Φ and initialize the watches */
3  foreach clause C ∈ Φ do
4      if C is unit on ℓ then
5          assign(ℓ, C)
6      else
7          installWatches(C)
8  while true do
9      C = UP()
       /* Unit propagation generated a falsified clause C */
10     if C ≠ NULL then
           /* If there is a conflict at decision level 0, Φ is UNSAT */
11         if dlevel == 0 then
12             return UNSAT
           /* Perform clause learning */
13         L = analyze(C)
14         installWatches(L)
           /* Non-chronological backtracking */
15         backtrack(level[L[1]])
16         assign(L[0], L)
17     else
           /* Solution found */
18         if there is no unassigned literal then
19             return SAT
           /* New decision */
20         ℓ = chooseLiteral()
21         dlevel = dlevel + 1
22         assign(ℓ, NULL)
```

Algorithm 12: The function DPLL uses to perform unit propagation, that returns a falsified clause if one is found, and otherwise returns *NULL*.

```
1 UP ()
2 while UPstack is not empty do
3     u = UPstack.pop()
4     foreach clause C ∈ watches[¬u] do
5         if C is unit on ℓ then
              /* The assign function will also push ℓ onto UPstack */
6             assign(ℓ, C)
7         if C is falsified then
8             return C
          /* Replace the watch ¬u with an unassigned or satisfied literal */
9         renewWatches(C)
```

Algorithm 13: The function DPLL uses to learn a 1-UIP clause from a conflict. The input C is a falsified clause. The function returns a learnt clause whose first literal belongs to the current decision level and whose other literals are sorted by decreasing decision level.

```
1 analyze (C)
2 L = C
3 while L contains two literals at level dlevel do
4     ℓ = a literal in L with maximum level
      /* Resolve L and the reason for ¬ℓ */
5     L = L∪ reason[¬ℓ] \{ℓ, ¬ℓ}
  /* Sort the literals in L by decreasing decision level */
6 return sort(L)
```

Bibliography

Teresa Alsinet, Felip Manyà, and Jordi Planes. Improved branch and bound algorithms for Max-SAT. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, 2003.

Teresa Alsinet, Felip Manyà, and Jordi Planes. A Max-SAT solver with lazy data structures. In *IBERAMIA*, pages 334–342, 2004.

Arne Andersson, Mattias Tenhunen, and Fredrik Ygge. Integer programming for combinatorial auction winner determination. In *Proceedings of the International Conference on MultiAgent Systems (ICMAS)*, pages 39–46, 2000.

Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables into problems with boolean variables. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 1–15, 2004.

Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 427–440, 2009.

Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial MaxSAT. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 3–8, 2010.

Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based

weighted MaxSAT solvers. In *Principles and Practice of Constraint Programming (CP)*, pages 86–101, 2012.

David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

Josep Argelich and Felip Manyà. Solving over-constrained problems with SAT technology. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 60–67, 2005.

Josep Argelich and Felip Manyà. Partial Max-SAT solvers with clause learning. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 28–40, 2007.

Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. The MaxSAT evaluations. 2007–2012. `http://www.maxsat.udl.cat`.

Josep Argelich, Daniel Le Berre, Inês Lynce, João Marques-Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. In *Proceedings of the First International Workshop on Logics for Component Configuration (LoCoCo)*, pages 11–22, 2010.

James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Proceedings of the International Conference on Practical Aspects of Declarative Languages (PADL)*, pages 174–186, 2005.

Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-boolean constraints into CNF. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 181–194, 2009.

E. Bensana, M. Lemaitre, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4:293–299, 1999.

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

María Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence (AI)*, 171:606–618, 2007.

Karthekeyan Chandrasekaran, Richard Karp, Erick Moreno-Centeno, and Santosh Vempala. Algorithms for implicit hitting set problems. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 614–629, 2011a.

Karthekeyan Chandrasekaran, Richard Karp, Erick Moreno-Centeno, and Santosh Vempala. Algorithms for implicit hitting set problems. *CoRR*, abs/1102.1472, 2011b.

Yibin Chen, Sean Safarpour, Andreas Veneris, and João Marques-Silva. Spatial and temporal design debug using partial MaxSAT. In *Proceedings of the ACM Great Lakes Symposium on VLSI*, pages 345–350, 2009.

Yibin Chen, Sean Safarpour, João Marques-Silva, and Andreas Veneris. Automated design debugging with maximum satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(11):1804–1817, 2010.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

M. C. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence (AI)*, 174(7-8):449–478, 2010.

Sylvain Darras, Gilles Dequen, Laure Devendeville, and Chu Min Li. On inconsistent clause-subsets for Max-SAT solving. In *Principles and Practice of Constraint Programming (CP)*, pages 225–240, 2007.

Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler

SAT instances. In *Principles and Practice of Constraint Programming (CP)*, pages 225–239, 2011.

Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, 2013.

Jessica Davies, Jeremy Cho, and Fahiem Bacchus. Using learnt clauses in MAXSAT. In *Principles and Practice of Constraint Programming (CP)*, pages 176–190, 2010.

M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

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

Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 252–265, 2006.

Alan Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 141–146, 2009.

Ana Graça, João Marques-Silva, Inês Lynce, and Arlindo Oliveira. Haplotype inference with pseudo-boolean optimization. *Annals of Operations Research*, 184:137–162, 2011.

Ana Graça, Inês Lynce, João Marques-Silva, and Arlindo L. Oliveira. Efficient and accurate haplotype inference by combining parsimony and pedigree information. In *Proceedings of the International Conference on Algebraic and Numeric Biology (ANB)*, pages 38–56, 2012.

Federico Heras and David Bañeres. The impact of Max-SAT resolution-based preprocessors on local search solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):89–126, 2010.

Federico Heras and João Marques-Silva. Read-once resolution for unsatisfiability-based Max-SAT algorithms. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 572–577, 2011.

Federico Heras, Javier Larrosa, and Albert Oliveras. MiniMaxSat: A new weighted Max-SAT solver. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 41–55, 2007.

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

Federico Heras, António Morgado, and João Marques-Silva. Core-guided binary search algorithms for maximum satisfiability. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 36–41, 2011.

Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively P-simulate general propositional resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 283–290, 2008.

John N. Hooker. Planning and scheduling by logic-based benders decomposition. *Operations Research*, 55(3):588–602, 2007.

Eric I. Hsu and Sheila A. McIlraith. Computing equivalent transformations for combinatorial optimization by branch-and-bound search. In *Proceedings of the Annual Symposium on Combinatorial Search (SOCS)*, pages 111–118, 2010.

David S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 38–49, 1973.

Farah Juma, Eric I. Hsu, and Sheila A. McIlraith. Exploiting MaxSAT for preference-based planning. In *Proceedings of the ICAPS-11 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (COPLAS)*, 2011.

Richard M. Karp. Implicit hitting set problems and multi-genome alignment. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, page 151, 2010.

Matthew Kitching and Fahiem Bacchus. Exploiting decomposition in constraint optimization problems. In *Principles and Practice of Constraint Programming (CP)*, pages 478–492, 2008.

Donald E. Knuth. Dancing links. In *Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, pages 187–214, 2000.

Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: a partial Max-SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:95–100, 2012.

Lukas Kroc, Ashish Sabharwal, and Bart Selman. Relaxed DPLL search for MaxSAT. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 447–452, 2009.

Adrian Kügel. Improved exact solver for the weighted Max-SAT problem. In *Workshop on the Pragmatics of SAT*, 2010.

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

Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, pages 66–76, 2000.

Chu Min Li, Felip Manyà, and Jordi Planes. Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In *Principles and Practice of Constraint Programming (CP)*, pages 403–414, 2005.

Chu Min Li, Felip Manyà, and Jordi Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2006.

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

Chu Min Li, Felip Manyà, Nouredine Ould Mohamedou, and Jordi Planes. Exploiting cycle structures in Max-SAT. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 467–480, 2009.

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

Han Lin, Kaile Su, and Chu Min Li. Within-problem learning for efficient lower bound computation in Max-SAT solving. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 351–356, 2008.

Vasco Manquinho, João Marques-Silva, and Jordi Planes. Algorithms for weighted boolean optimization. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 495–508, 2009.

João Marques-Silva and Vasco M. Manquinho. Towards more effective unsatisfiability-based maximum satisfiability algorithms. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 225–230, 2008.

João Marques-Silva and Jordi Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, abs/0712.1097, 2007.

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

Ruben Martins, Vasco Manquinho, and Inês Lynce. Clause sharing in parallel MaxSAT. In *Proceedings of the International Conference on Learning and Intelligent Optimization (LION)*, pages 455–460, 2012a.

Ruben Martins, Vasco M. Manquinho, and Inês Lynce. On partitioning for maximum satisfiability. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 913–914, 2012b.

Antonio Morgado, Federico Heras, and João Marques-Silva. Improvements to core-guided binary search for MaxSAT. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 284–297, Berlin, Heidelberg, 2012.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the Annual Design Automation Conference (DAC)*, pages 530–535, 2001.

Christian Muise, Sheila A. McIlraith, and J. Christopher Beck. Optimization of partial-order plans via MAXSAT. In *Proceedings of the ICAPS-11 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (COPLAS)*, pages 31–38, 2011.

Christos H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *Journal of Computer and System Sciences*, 37(1):2–13, 1988.

Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

James D. Park. Using weighted MAX-SAT engines to solve MPE. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 682–687, 2002.

Thierry Petit, Christian Bessière, and Jean-Charles Régin. A general conflict-set based framework for partial constraint satisfaction. In *Proceedings of the International Workshop on Soft Constraints*, 2003.

Knot Pipatsrisawat and Adnan Darwiche. Clone: solving weighted Max-SAT in a reduced search space. In *Proceedings of the Australian Joint Conference on Advances in Artificial Intelligence (AUS-AI)*, pages 223–233, 2007.

Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers with restarts. In *Principles and Practice of Constraint Programming (CP)*, pages 654–668, 2009.

Reza Ramezani and Sayed Rasoul Mousavi. Description of the IUT-RR solvers. In *2012 Maxsat Evaluation*, 2012. `http://maxsat.ia.udl.cat:81/12/solvers/index.html`.

Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. Partial weighted MaxSAT for optimal planning. In *Proceedings of the Pacific Rim International Conference on Trends in Artificial Intelligence (PRICAI)*, pages 231–243, 2010.

Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 174–187, 2011.

SHARCNET. The shared hierarchical academic research computing network. `www.sharcnet.ca`, 2013.

João P. Marques Silva and Inês Lynce. On improving MUS extraction algorithms. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 159–173, 2011.

Dawn M. Strickland, Earl Barnes, and Joel S. Sokol. Optimal protein structure alignment using maximum cliques. *Operations Research*, 53:389–402, 2005.

Dave Tompkins and Holger Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT)*, pages 306–320, 2004.

Vijay Vazirani. Approximation algorithms. Springer-Verlag, 2001.

Richard J. Wallace and Eugene C. Freuder. Comparative studies of constraint satisfaction and davis-putnam algorithms for maximum satisfiability problems. *The Second DIMACS Challenge: Cliques, Coloring and Satisfiability*, 26:587–615, 1996.

Karsten Weihe. Covering trains by stations or the power of data reduction. In *Proceedings of the Workshop on Algorithms and Experiments (ALEX)*, pages 1–8, 1998.

Laurence A. Wolsey. *Integer Programming*. Whiley, 1998.

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

Hui Xu, Rob A. Rutenbar, and Karem Sakallah. sub-SAT: a formulation for relaxed boolean satisfiability with applications in routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):814–820, 2003.

Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 210–216, 2010.

Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence (AI)*, 171:107–143, 2007.

Hantao Zhang, Haiou Shen, and Felip Manyà. Exact algorithms for MAX-SAT. *Electronic Notes in Theoretical Computer Science*, 86(1):190–203, 2003.