Solving Weighted Max-SAT Problems in a Reduced Search Space: A Performance Analysis*

Knot Pipatsrisawat, Akop Palyan, Mark Chavira, Arthur Choi, and Adnan Darwiche
Correspondence to Knot Pipatsrisawat (thammakn@cs.ucla.edu)

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095 USA

Abstract

We analyze, in this work, the performance of a recently introduced weighted Max-SAT solver, Clone, in the Max-SAT evaluation 2007. Clone utilizes a novel bound computation based on formula compilation that allows it to search in a reduced search space. We study how additional techniques from the SAT and Max-SAT literature affect the performance of Clone on problems from the evaluation. We then perform further investigations on factors that may affect the performance of leading Max-SAT solvers. We empirically identify two properties of weighted Max-SAT problems that can be used to adjust the difficulty level of the problems with respect to the considered solvers.

1. Introduction and Background

The maximum satisfiability problem (Max-SAT) is one of the optimization counterparts of the Boolean satisfiability problem (SAT). In Max-SAT, given a Boolean formula in conjunctive normal form (CNF), we want to determine the maximum number of clauses that can be satisfied by any complete assignment, where a clause is a disjunction of literals and a literal is simply a variable or its negation. Recently, the study of Max-SAT has been growing in popularity, as demonstrated by the quickly increasing number of Max-SAT solvers [2, 3, 4, 5, 6, 7]. The Max-SAT problem has also been used as a model for many applications in areas such as databases [8], FPGA routing [9], automatic scheduling [10], and genotype analysis [11]. The annual Max-SAT evaluation has played an important role in advancing this field of study [12, 13].

Two important variations of the Max-SAT problem are the weighted Max-SAT and the partial Max-SAT problems. The *weighted* Max-SAT problem is the Max-SAT problem, in which each clause is assigned a positive weight. The objective of this problem is to maximize the sum of weights of satisfied clauses by any assignment. The *partial* Max-SAT problem is the Max-SAT problem, in which some clauses cannot be left falsified by any solution. In practice, a clause that cannot be falsified is represented by a clause with a sufficiently large weight. The combination of both variations is called the *weighted partial* Max-SAT problem.

*This work extends our previous work in [1]

For the rest of this paper, we use the term Max-SAT to refer to any variation of the Max-SAT problem.

There are two main approaches used by contemporary exact Max-SAT solvers: the satisfiability-based approach and the branch-and-bound approach. The former converts each Max-SAT problem with different hypothesized maximum weights into multiple SAT problems and uses a SAT solver to solve these SAT problems to determine the actual solution. Examples of this type of solver are ChaffBS, ChaffLS [14] and SAT4J-MaxSAT [15]. The second approach, which seems to dominate in terms of performance based on recent Max-SAT evaluations [12, 13], utilizes a depth-first branch-and-bound search in the space of possible assignments. An evaluation function which computes a bound is applied at each search node to determine any pruning opportunity.

The methods used to compute bounds vary among branch-and-bound solvers and often give rise to difference in performance. Toolbar utilizes local consistencies to aid bound computations [4, 16]. Lazy, MaxSatz, PMS, LB-SAT, and MiniMaxSAT compute bounds using some variations of unit propagation and disjoint component detection [6, 17, 18, 7, 3, 2]. Moreover, solvers such as MiniMaxSAT and PMS also use Max-SAT inference rules to improve bound quality.[1]

Our solver, Clone, uses a completely different approach for computing bounds. Clone compiles a relaxed version of the Max-SAT problem into a tractable form and computes bounds from the compiled formula. This approach allows our solver to better take advantage of problem structure. Moreover, it can be thought of as an approach that combines search and compilation together. Note that the Max-SAT solver Sr(w) by Ramírez and Geffner [5] uses the same approach for bound computation. Both solvers were developed independently and both participated in the Max-SAT evaluation 2007 [13].

In this work, we experimented with additional techniques for improving our Max-SAT solver and evaluated their impact on problems from the Max-SAT evaluation. We analyzed the performance of our solver and performed further experiments that revealed a class of problems on which our solver significantly outperformed other Max-SAT solvers. This result demonstrates the benefits of our approach. Our investigation also led us to identify some properties of Max-SAT problems that can be used to indicate their difficulty. We report empirical results that show how these properties can be manipulated for different difficulty levels of Max-SAT problems.

In the next section, we discuss the preprocessor of Clone and our approach for computing bounds. In Section 3, we describe the search component and the inference techniques used in Clone. Evaluation of our solver on problems from the Max-SAT evaluation is presented in Section 4. In Section 5, we carefully analyze the performance of our solver and discuss some properties of problems used in the evaluation. In Section 6, we present a series of results from our additional investigations that allow us to identify some properties of weighted Max-SAT problems that are good indicators of solvers' performance. Finally, we conclude with some remarks in Section 7.

1 Local consistency and Max-SAT inference are two highly-related concepts (see [19]).

2. Bound Computation

In the literature, the Max-SAT problem is often viewed as the problem of minimizing the costs (or weights) of falsified clauses of any assignment. We will follow this interpretation and use the term *cost* to refer to the sum of weights of clauses that are not satisfied. Moreover, we will use UB (upper bound) to denote the best cost of any complete assignment found so far and LB (lower bound) to denote the lower bound on the cost of any assignment that extends the current partial assignment. Branch-and-bound search algorithm can prune all children of a node whenever $LB \geq UB$.

To compute lower bounds, we take advantage of a tractable language called deterministic decomposable negation normal form (d-DNNF) [20, 21]. The key property of d-DNNF that we utilize here is the fact that, for each conjunction in a d-DNNF formula, the conjuncts share no variable. This property is called *decomposability* [22]. Many useful queries can be answered about sentences in d-DNNF in time linear in the size of these sentences. One of these queries is (weighted) minimum cardinality, which is similar to Max-SAT, except that weights are associated with variables instead of clauses. Our approach is indeed based on reducing Max-SAT on the given CNF to minimum cardinality on a d-DNNF equivalent of the CNF. If this compilation is successful, the Max-SAT problem is solved immediately since minimum cardinality can be solved in time linear in the d-DNNF size. Unfortunately, however, the compilation process is often difficult. Our solution to this problem is then to compile a relaxation of the original CNF. The relaxed CNF is generated carefully to make the compilation process feasible. The price we pay for this relaxation is that solving minimum cardinality on the resulting d-DNNF (of the relaxed CNF) will give lower bounds instead of exact solutions. Our approach is then to use these lower bounds for pruning in our branch-and-bound search.

We show how a Max-SAT problem can be reduced to a minimum cardinality problem in Section 2.1. We will then discuss problem relaxation in Section 2.2, followed by compilation in Section 2.3. A method for computing bounds from the compiled formula is discussed in Section 2.4.

2.1 Reducing Max-SAT to Minimum Cardinality

Given a CNF formula and a cost for each literal of the formula, the weighted minimum cardinality problem asks for a satisfying assignment that costs the least. This problem is also known as the MinCostSAT problem [9]. The cost of an assignment is the sum of the costs of all literals that it sets to true. To reduce a Max-SAT problem into a minimum cardinality problem, we introduce a distinct selector variable to each clause of the Max-SAT problem and assign the clause's cost to the positive literal of the selector variable [23]. All other literals are assigned zero cost. For example, the clause $C = (a \vee b \vee c)$ becomes $C' = (s \vee a \vee b \vee c)$ after the selector variable s is added. If C originally had cost w associated with it, then w is assigned to s and any assignment that set $s = \textbf{true}$ will incur this cost. After this conversion, the formula will be trivially satisfiable, because every clause contains a distinct selector variable. Nevertheless, finding a satisfying assignment with the lowest cost is not easy. The minimum cardinality problem is NP-hard for CNF formulas. However, it can be solved efficiently once we have the formula in d-DNNF. Any solution

to this problem can be converted back to a solution for the original Max-SAT problem by ignoring assignments of the selector variables.

At this point, we are almost ready to compile the CNF formula. The only remaining issue is the time complexity of the compilation, which is, in the worst case, exponential in the treewidth of the constraint graph [24] of the CNF formula. The treewidth of a graph is a theoretic parameter, which measures the extent to which a graph resembles a tree (the lower the treewidth, the more tree-like) [25]. In most cases, straight compilation will be impractical. As a result, we need to relax the formula to lower its treewidth.

2.2 Problem Relaxation by Variable Splitting

The approach we use to relax the problem is called *variable splitting*, which was inspired by the work of Choi *et al* in [26]. In general, splitting a variable v involves introducing new variables for all but one occurrence of v in the original CNF formula.[2] For example, splitting a in the CNF $(a \vee b) \wedge (\neg a \vee c) \wedge (a \vee d) \wedge (b \vee \neg c) \wedge (c \vee \neg d)$ results in the formula $(a \vee b) \wedge (\neg a_1 \vee c) \wedge (a_2 \vee d) \wedge (b \vee \neg c) \wedge (c \vee \neg d)$. In this case, a is called the *split variable*. The new variables (a_1 and a_2 in this case) are called the *clones* of the split variable. Figure 1 illustrates the constraint graph of the above CNF before and after the split.

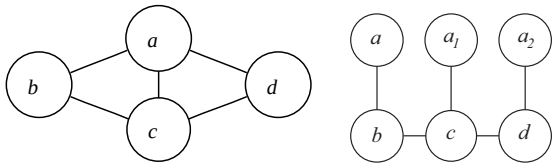

Figure 1: (left) The constraint graph of $(a \vee b) \wedge (\neg a \vee c) \wedge (a \vee d) \wedge (b \vee \neg c) \wedge (c \vee \neg d)$. (right) The constraint graph after splitting a. The treewidth is reduced from 2 to 1.

After splitting, the resulting problem becomes a relaxation of the original problem, because any assignment in the original problem has an assignment in the split problem with the same cost. Such an assignment can be obtained by setting the value of every clone according to its split variable. Therefore, the lowest cost of any split formula is a lower bound of the lowest cost of the original formula. The strategy we use for selecting split variables is the same as the one described in [26]. Identifying variables to split is closely related to the problem of finding a loop cutset (or cycle cutset) [27, 28], except that we do not necessarily insist on splitting variables until the constraint graph becomes a tree.

2.3 CNF to d-DNNF Compilation

Once the problem has a sufficiently low treewidth, it can be practically compiled. The process of compiling a CNF formula into a d-DNNF formula is performed by a program called C2D [21, 29]. C2D takes a CNF formula as input and produces an equivalent formula in d-DNNF. The output formula is fed to the search engine and will be used for later bound computations.

2 This type of splitting is called full splitting. While other degrees of splitting are possible, we focus our attention only to this method.

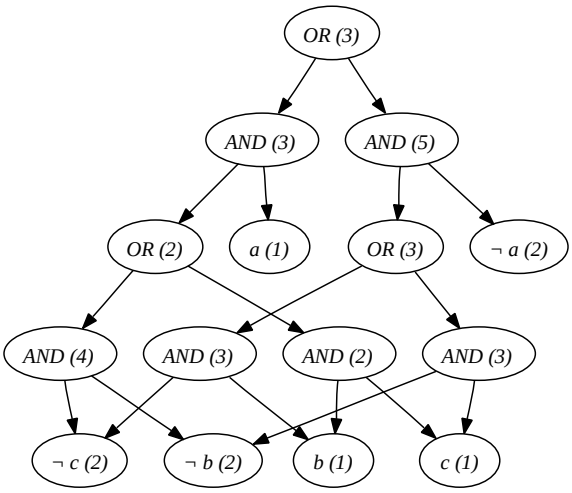

Figure 2: The DAG of the d-DNNF formula $(a \wedge ((b \wedge c) \vee (\neg b \wedge \neg c))) \vee (\neg a \wedge ((b \vee \neg c) \vee (\neg b \vee c)))$. Each node in this graph is also labeled with the value used to compute the minimum cardinality of the root.

2.4 Computing Bounds from d-DNNF

Every d-DNNF formula can be represented as a rooted DAG. Each node in the DAG is either a Boolean constant, a literal, or a logical operator (conjunction or disjunction). The root of the DAG corresponds to the formula. For example, consider the DAG of a d-DNNF formula $(a \wedge ((b \wedge c) \vee (\neg b \wedge \neg c))) \vee (\neg a \wedge ((b \vee \neg c) \vee (\neg b \vee c)))$ in Figure 2. In this figure, the cost of every positive literal is set to 1 and the cost of every negative literal is set to 2. The minimum cardinality of the formula is simply the value of the root node, which is defined recursively as [23]:

1. The value of a literal node is the value of the literal
2. The value of an AND node is the sum of the values of all its children
3. The value of an OR node is the minimum of the values of its children

Note that Step 2 is possible because the formula satisfies decomposability. If the formula is a relaxed formula, then the computed minimum cardinality becomes a lower bound of the minimum cardinality of the formula before relaxation (hence a lower bound of the optimal cost of the original Max-SAT problem). The d-DNNF formula can also be efficiently conditioned on any partial or complete assignment of its variables. Conditioning only affects the values of the nodes whose literals are set to false. False literals can no longer contribute to any solution of the problem. Hence, the values of such nodes are set to ∞, which may in turn affect the values of their parents or ancestors. The resulting bound computed from the conditioned formula will be a lower bound of the optimal cost of any assignment that extends the conditioning assignment.

In the branch-and-bound search, we need to compute a bound at every search node under the partial assignment at that node. To make this process efficient, bounds are

computed incrementally as follows. For each node, we stored the most recent computed value and maintain a *touched* bit. A literal node is touched if and only if its truth value has changed since the last bound computation. When a new bound needs to be computed, we traverse the formula in a bottom-up manner and only re-evaluate touched nodes, because the stored values of untouched nodes are still correct. Once a touched node is evaluated, we reset its touched bit (to untouched), mark its parents as touched and continue. As a result, we need to set touched bits of different literals appropriately as we explore the search space. A literal is marked touched whenever its value changes; when it is assigned a value or backtracked past.

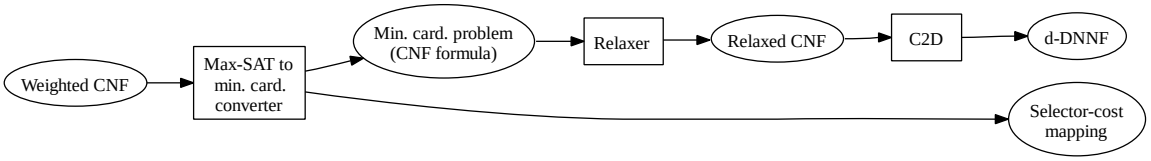

Figure 3: A system diagram of Clone's preprocessor.

We close this section by summarizing in Figure 3 the relationship between different parts of Clone's preprocessor. The input is a Max-SAT problem (weighted CNF formula). The preprocessor produces a d-DNNF formula and information about the costs of selector variables. These outputs are passed on to the branch-and-bound search engine, which we describe in the next section.

3. Search and Inference

The next component of Clone is the branch-and-bound search engine. The engine uses bounds computed from the d-DNNF formula for pruning. The search algorithm only branches on variables in the original Max-SAT problem (as opposed to clones or selectors). Every time a split variable is assigned a value, the algorithm ensures that all its clones are set to the same value. Otherwise, a solution in this search space may be meaningless in the original problem.

Next, we describe some techniques that are utilized in Clone to improve the efficiency of the search. Some of these techniques require access to the clauses in the original Max-SAT problem. Hence, although we compile the CNF formula into a d-DNNF, the original formula (weighted CNF) is also given to the search engine.

3.1 Reducing the Size of the Search Space

Given our method for computing bounds, it is possible to reduce the size of the search space that the algorithm needs to explore. Since the relaxed problem differs from the original problem only on split variables and their clones, as soon as every split variable (and its clones) is set to a value, the two problems become identical under the current assignment (recall that we ensure that every clone is set according to its split variable). Therefore, the bound computed from the d-DNNF formula at this point must be the exact Max-SAT optimal cost of the original problem under the current assignment.

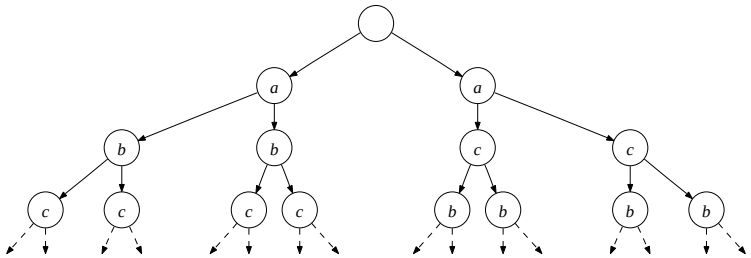

Figure 4: An example search space with each node labeled with the last branch variable.

For example, consider the search space of a problem with 3 split variables–a, b, and c–in Figure 4. The bound computed at each node is a lower bound of the optimal cost of any complete assignment that extends the assignment at the node. However, once all three split variables are instantiated, the bound becomes exact. Therefore, there is no need to visit any node below depth 3 in this search tree, regardless of the number of other variables. This realization suggests a natural way of reducing the size of the search space; we only need to search in the space of assignments of split variables. A problem with a large treewidth is likely to require many variables to be split, yielding a small reduction in the size of the search space. On the other extreme, a problem that can be compiled without any splitting can be solved without any search, because the bound computed without any conditioned variable is already equal to the exact optimal cost.

For the remaining of this section (3.2-3.6), we will only discuss techniques that deal with the original weighted CNF formula. Therefore, *a clause* always refer to an original clause of the problem (as opposed to that in the minimum cardinality or the relaxed problem).

3.2 Unit Propagation

Unit propagation is a predominant inference rule in SAT solvers. The rule states that whenever a unit clause (ℓ) exists in the formula, the literal ℓ must essentially be set to **true**. However, this rule cannot be applied to all clauses in a Max-SAT problem, because not every clause needs to be satisfied. Unit propagation can be applied to a unit clause C only if C has a sufficiently large weight ($\geq UB$) to be considered mandatory under the current assignment. These mandatory clauses are called *hard clauses* and applications of unit propagation on hard clauses generate *hard implications*.

A *soft clause*, on the other hand, is a clause with weight less than UB. A soft clause may become a hard clause over time as the value of UB decreases. Whenever a soft clause becomes unit over literal ℓ, we add the cost of the clause to the cost of ℓ.[3] This represents the fact that falsifying ℓ will automatically falsify the clause and will incur the associated cost. As soon as the cost of a literal is greater than or equal to UB, the literal is implied to **true**. This is called a *soft implication*, which is similar to the application of node consistency [30]. Moreover, as soon as a soft clause becomes unit over ℓ, it is added to the list of unit soft clauses of ℓ. This list will be used later when Clone tries to derive a conflict clause. In our

3 This cost is used only for the implementation of unit propagation and not for bound computation.

implementation, the two-watched literal scheme [31] is used to efficiently implement unit propagation.

3.3 Dealing with Hard Conflicts

A *hard conflict* is a conflict in which a hard clause gets violated. This type of conflict may arise during unit propagation and presents the solver with a pruning opportunity. Clone employs non-chronological backtracking, which is common in CSP and SAT [32, 33, 34]. Conflict analysis as described in [35] is performed upon every hard conflict. As a result, an earlier search level is identified as the target of the subsequent backtrack. This form of backtracking allows Clone to prune the search tree efficiently.

In addition to non-chronological backtracking, Clone also employs an important technique from SAT called conflict clause learning. Upon each conflict, the solver derives a conflict clause using the 1-UIP scheme as described in [36]. A conflict clause helps prune some parts of the search tree that do not contain the optimal solution. To derive a conflict clause, the solver needs to construct an implication graph associated with the conflict [36]. During this process, the reasons of some (hard and soft) implications need to be analyzed. The reason of a hard implication is the hard clause in which the implied literal becomes unit. The reason of a soft implication, on the other hand, needs to be recovered by traversing the list of soft clauses in which the implied literal appears alone after simplification (recall that we maintain a list of unit soft clauses for each literal). The disjunction of these clauses is a reason of the soft implication. For example, if soft implication ℓ appears in clauses $(\ell \vee x), (\ell \vee y \vee z), (\ell \vee w)$ and x, y, z, w are all set to false, then $(\ell \vee w \vee x \vee y \vee z)$ is a reason of ℓ. Once the conflict clause is learned by the solver, unit propagation will guarantee that the solver will not run into the same conflict again. Other Max-SAT solvers that learn conflict clauses upon hard conflicts include MiniMaxSAT [2], PMS [7], and Sr(w) [5].

3.4 Dealing with Soft Conflicts

Recall that whenever the current lower bound is greater than or equal to the current upper bound ($LB \geq UB$), the solver can prune the current branch of the search. This situation is called a *soft conflict*, because no hard clause is actually violated. Traditionally, a branch-and-bound search algorithm backtracks chronologically (flip the most recent unflipped decision) when a soft conflict is found. The standard SAT techniques for dealing with conflicts do not apply directly to this type of conflict, because no hard clause is violated in this case. However, in some cases, we could efficiently construct a violated hard clause from a soft conflict.

Upon each soft conflict, clearly $LB \geq UB$. Moreover, some soft clauses may already be falsified by the current partial assignment. However, it is not necessary that the sum S of the costs of the violated soft clauses be greater than or equal to UB. This is simply because the method used for computing lower bounds may be able to implicitly identify certain clauses that can never be satisfied at the same time, even when their truth values are still in question. In any case, whenever $S \geq UB$, a violated hard clause can be artificially constructed. In particular, let $\{C_1, C_2, ..., C_k\}$ be a set of violated soft clauses whose sum of costs exceeds (or equal) UB. Then, $C = \vee_{i=1}^{k} C_i$ is a violated hard clause. The clause C simply makes explicit the fact that $C_1, ..., C_k$ cannot all be false at the same time.

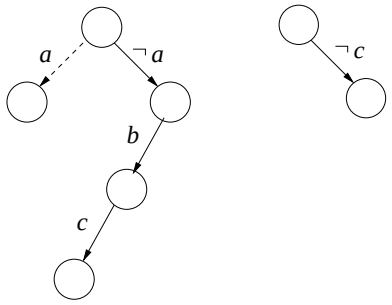

Figure 5: (a) A search tree in which a has been flipped by chronological backtracking. (b) The search tree after non-chronological backtracking and learning conflict clause ($\neg c$).

To perform conflict analysis on a soft conflict, we just need to find a minimal set of violated soft clauses whose sum of costs is greater than UB. The disjunction of the literals of these clauses is a violated hard clause, which can be used by the standard conflict analysis algorithm to generate a conflict clause, which is a hard clause, and to compute a level to non-chronologically backtrack to. In other cases, in which $S < UB$, Clone simply backtracks chronologically without learning any clause.

Note that this is not the first time a learning scheme for soft conflicts is proposed. A similar scheme was proposed in the partial Max-SAT solver PMS [7]. The learning scheme of PMS utilizes Max-SAT resolution rule [19] and, according to [7], has not been coupled with non-chronological backtracking. Another solver that attempts to learn from soft conflicts is Sr(w) [5]. It generates a violated hard clause by traversing the compiled formula and applies standard conflict analysis procedure to the clause.

3.5 Avoiding Work Repetition

Both chronological and non-chronological backtracking are employed by Clone. The former is used when the solver encounters a soft conflict for which no violated hard clause can be easily constructed. In this case, no new clause is learned by the solver. Non-chronological backtracking is used for the other pruning opportunities and a new conflict clause is learned every time.

Combining these two types of backtracking naively may lead the solver to repeat a lot of work in some situations. For example, consider the search tree in Figure 5 (a). In this search tree, the variable a was flipped to $\neg a$ by a chronological backtrack, which indicates that the whole subtree under $a = \textbf{true}$ had been exhausted. After a few more decisions (b and c, in this example), the solver may run into a hard conflict and derive a conflict clause ($\neg c$). As a result, non-chronological backtracking will erase every assignment and set $c = \textbf{false}$ at the very top level (level 0). The search tree after backtracking is shown in Figure 5 (b). The assignment $a = \textbf{false}$ had been erased in the process and there is no way to recover the fact that the branch $a = \textbf{true}$ had been fully explored. To the best of our knowledge, this problem had never been dealt with before in any Max-SAT solver that employs both chronological and non–chronological backtracking.

To solve this problem, every time Clone non-chronologically backtracks past any chronologically flipped assignment, it records a *blocking clause*, which is a hard clause that captures that information. More formally, let ℓ be a current literal assignment that was flipped by

a chronological backtrack at level k. If the solver non-chronologically backtracks past ℓ, it will learn $C = (\ell \vee \neg d_1 \vee \neg d_2 \vee ... \vee \neg d_{k-1})$, where d_i is the decision assignment at level i. In the above example, $(\neg a)$ would be learned since it was the very first decision in the search tree. The use of blocking clauses here is similar to those employed in other contexts such as SAT-based model checking [37].

As we can see, Clone constantly adds new clauses to the formula as it is solving the problem. These clauses could represent a considerable memory overhead and must be deleted to ensure that Clone does not exhaust the memory too quickly. As a result, Clone periodically deletes inactive learned clauses from the formula. The activity of each clause is heuristically determined by its participation in recent conflict analysis as normally done in DPLL-based SAT solvers [38].

3.6 Other Improvements

Apart from the above techniques, Clone computes an initial upper bound by calling Maxwalksat [39], which is a local search solver. Clone uses the best cost found by Maxwalksat after a fixed number of trials as the initial upper bound.

Clone utilizes a dynamic variable ordering heuristic that is a combination of the two-sided weighed Jeroslow-Wang (JW) [40] and VSIDS [31] heuristics. In our implementation of weighted JW, we prefer the literal with the highest weight. The weight of a literal ℓ, in the simplified formula, is defined to be $\sum_{\ell \in C} w(C)2^{|C|}$, where $w(C)$ is the weight of clause C. In general, JW performs well on weighted problems as it takes into account both clause lengths and clause weights. However, we found that VSIDS tends to outperform JW dramatically on problems with many hard clauses, which usually result in many hard conflicts experienced by the solver. A similar observation was reported in [2]. Clone always starts by using JW heuristic and dynamically switches to VSIDS if hard conflicts are detected sufficiently often.

4. Experimental Results

In this section, we present experimental results that compare different versions of Clone in order to analyze the contributions of different techniques. We focus on non-random problems from the weighted and unweighted partial Max-SAT categories of the latest Max-SAT evaluation [13].

The version of Clone that participated in the Max-SAT evaluation 2007 is a basic solver that did not incorporate most techniques described in Section 3. It had no initial upper bound computation and only used the JW heuristic for ordering variables. It also did not have any technique for handling soft conflicts. In this experiment, we consider the following versions of Clone.

1. The version of Clone that participated in the Max-SAT evaluation 2007 (S1).
2. Version 1 + initial upper bound computation using Maxwalksat (S2).
3. Version 2 + learning from soft conflicts and avoiding work repetition (S3).
4. Version 3 + a variable ordering heuristic that combines JW with VSIDS (S4).

Family	Total	Solved problems (median running time (s))				
		S1	S2	S3	S4	S5
auction paths	88	88 (2.65)	88 (2.23)	88 (2.01)	88 (1.98)	88 (1.16)
auction regions	84	84 (7.97)	84 (6.47)	84 (5.70)	84 (5.83)	84 (0.24)
auction scheduling	84	77 (19.89)	77 (18.95)	78 (15.09)	78 (15.09)	84 (2.92)
pseudo factor	186	186 (2.26)	186 (1.9)	186 (1.81)	186 (1.76)	186 (0.086)
pseudo miplib	16	5 (0.62)	5 (0.36)	5 (0.36)	5 (0.36)	5 (0.10)
qcp	25	0 (-)	0 (-)	23 (11.87)	24 (12.28)	20 (0.36)
planning	71	71 (7.94)	71 (7.45)	71 (6.93)	71 (6.39)	71 (0.051)
spot5 dir	21	6 (0.83)	6 (0.5)	6 (0.6)	6 (0.55)	3 (0.49)
spot5 log	21	6 (1.19)	6 (0.78)	6 (0.69)	6 (0.67)	4 (2.93)
Total	596	523	523	547	548	545

Table 1: The number of solved weighted partial Max-SAT problems from the Max-SAT evaluation 2007. The median running time (on solved problems) are shown in parentheses.

Family	Total	Solved problems (median running time (s))				
		S1	S2	S3	S4	S5
maxclique random	96	81 (11.72)	80 (12.29)	81 (9.40)	80 (8.98)	96 (0.12)
maxclique structured	62	17 (16.70)	19 (43.26)	18 (56.44)	17 (14.67)	36 (3.76)
maxone 3sat	80	54 (58.83)	59 (44.68)	62 (39.06)	59 (40.03)	80 (4.21)
maxone structured	60	31 (17.14)	31 (16.14)	37 (20.17)	36 (19.65)	60 (2.87)
pseudo garden	7	5 (0.11)	5 (0.1)	5 (0.1)	5 (0.1)	5 (0.088)
pseudo logic-synthesis	17	0 (0.0)	1 (265.90)	1 (1072.98)	1 (905.87)	2 (177.03)
pseudo primes-dimacs-cnf	148	104 (3.21)	106 (2.11)	103 (2.06)	103 (2.01)	107 (0.030)
pseudo routing	15	5 (4.72)	5 (5.51)	5 (6.794)	7 (4.22)	14 (37.39)
maxcsp dense-loose	20	1 (776.41)	1 (622.10)	1 (723.32)	1 (568.73)	20 (0.18)
maxcsp dense-tight	20	20 (19.84)	20 (10.18)	20 (10.67)	20 (10.59)	20 (5.01)
maxcsp sparse-loose	20	14 (18.70)	16 (21.68)	15 (16.39)	17 (23.14)	20 (0.027)
maxcsp sparse-tight	20	20 (6.97)	20 (4.31)	20 (4.47)	20 (4.57)	20 (0.017)
wqueens	7	4 (1.56)	4 (1.93)	5 (1.72)	5 (1.69)	7 (0.18)
Total	572	356	367	373	371	487

Table 2: The number of solved partial Max-SAT problems from the Max-SAT evaluation 2007. The median running time (on solved problems) are shown in parentheses.

In all experiments, Clone relaxes every Max-SAT problem until its treewidth is less than or equal to 8, which we found to be empirically optimal on the large set of problems.

In what follows, we also report the performance of MiniMaxSAT (S5) on the considered set of problems as a point of reference. MiniMaxSAT is one of the most successful solvers in the evaluation. It outperformed others in the weighted and unweighted partial Max-SAT categories. All experiments were run on two Intel Pentium 4 machines, each with 3.79GHz and 4GB of RAM. The time-out is 1,200 seconds per problem.

Tables 1 and 2 show the performance of the solvers on the weighted partial Max-SAT and partial Max-SAT categories, respectively. In each of the five rightmost columns, the number

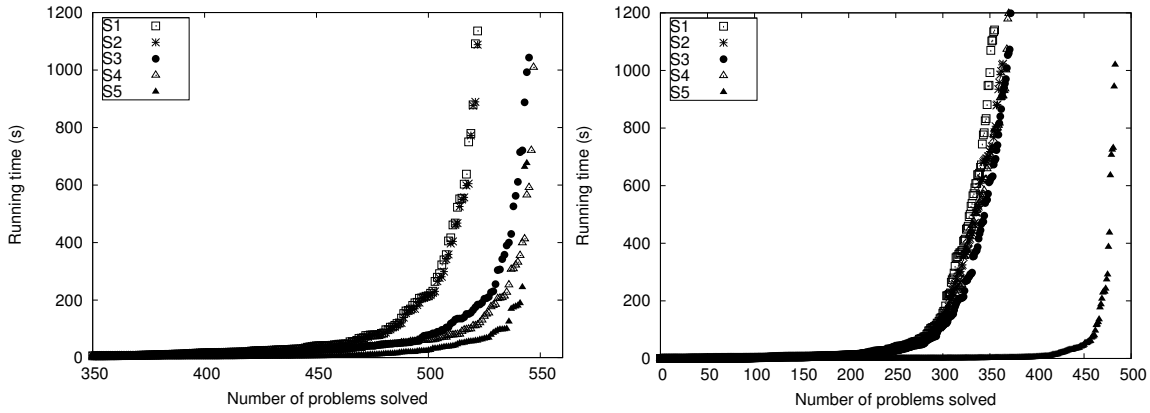

Figure 6: (a) Running time profile of all solvers on weighted partial Max-SAT problems and (b) on partial Max-SAT problems.

of solved problems are shown, with the median running time in parentheses. Figure 6 shows the performance profiles of all solvers on these two categories of problem. The results from both tables and the plots confirm that the additional techniques utilized by Clone considerably boost the solver's performance. In the weighted partial Max-SAT category, although using a local search solver did not allow Clone to solve more problems, it tends to reduce the running time on these problems, as indicated by lower median running time. Soft conflict handling technique and the new ordering heuristic, on the other hand, are very effective, allowing 25 more problems to be solved by Clone.

The main difference in performance of different versions of Clone lies in the QCP family. A closer look at this set of problems reveals something interesting. Let the term *soft weight* refer to the weight of a soft clause. Each problem in the QCP family contains a set of hard clauses and a set of soft clauses. There are only two small, distinct soft weights in each QCP problem. For most problems, it turns out that the local search algorithm used by Clone is able to find good upper bounds. As a result, most soft clauses turn into hard clauses, rendering the Max-SAT problem very similar to a normal SAT problem. When Clone solves this type of problem, it runs into many hard conflicts and performs non-chronological backtracking often. As a result, it benefited significantly from learning upon soft conflicts and from the technique for avoiding work repetition. The use of VSIDS heuristic also lowered the running time and allowed one additional problem to be solved. Notice that a good initial upper bound alone is not sufficient to allow Clone to solve problems in this family. It is interesting to note that SAT4Jmaxsat [15], which utilizes a SAT-based approach, performs relatively well on this family in the Max-SAT evaluation (only second to MiniMaxSAT) [41].

Overall, even though the best version of Clone solves 3 more problems than MiniMaxSAT, the running time of Clone tends to be larger than that of MiniMaxSAT. Note that, because of the modular structure of our system, overhead incurred due to file I/O is significant for easy problems.

In the partial Max-SAT category, the same trend, though more limited, can be seen in the performance of Clone. Clone with soft conflict handling and new ordering heuristic solves 15 problems more than the Max-SAT evaluation version, while the version without the new ordering heuristic solves 17 problems more than the base version. The improvement scatters throughout many problem families. In this category, however, even the best version of Clone still performs considerably worse than MiniMaxSAT. We expected Clone to perform relatively poorly on problems in which soft clauses are unweighted. The reason for this is because Clone is yet to incorporate any technique that takes advantage of this additional information (that all soft clauses have the same weight).

5. Further Analysis of Results

In this section, we performed further analysis on the results presented in the previous section in order to understand why Clone performed well or poorly on some problems. The technique used to compute bounds in Clone makes the solver sensitive to the treewidth of the problem. In particular, problems with high treewidth are likely to require many variables to be split, while problems with low treewidth may require none or only a few split variables. A large number of split variables has two adverse effects on Clone. First, it makes the size of Clone's search space large, because Clone searches in the space of split variables' assignments. Second, a large number of split variables means that the relaxed problem could be very different from the original problem. Hence, bounds computed from the compiled formula tend to be rather loose and, thus, contribute little to pruning.

To quantify this, we measured the treewidths of all problems used in our experiments. Since computing the treewidth of a graph is intractable in general, we present here only an upper bound of the treewidth of each problem. This upper bound is computed using the min-fill and min-size heuristics [42, 43]. In our future discussion, we will use the term *computed treewidth* to refer to this treewidth upper bound. Tables 3 and 4 report the computed treewidth information of all problem families. The first column in each table is the family name. The second column is the average computed treewidth of the problems in the family. The remaining columns, which are labeled with versions of Clone, contain the average computed treewidths of the problems that each solver could solve in the family.

In almost all families, all versions of Clone found the problems with low treewidths easier to solve. This is reflected by the fact that the average treewidth of solved problems are (much) smaller than the average treewidth of all problems in each family. For example, the average treewidth of the family "pseudo miplib" is 183, while the average treewidth of those solved by Clone is only 13. Similar patterns can also be found in families "spot5 dir", "spot5 log", "maxclique structured", "pseudo garden", and "wqueens". On the other end, families with sufficiently low treewidths tend to be solved almost completely or completely by Clone. Examples of such families are "auction paths", "pseudo factor", "maxcsp dense-tight","maxcsp sparse-tight", or even "maxcsp sparse-loose". Notice that Clone was able to solve some problems with high treewidths in some families. Such results are possible, as other factors, such as problem size and weights, need to be considered carefully when comparing problems with different structures.

Prior to this experiment, we did not know whether MiniMaxSAT was sensitive to problem structure or not. We expected MiniMaxSAT, however, to be less sensitive (if at all) to

Family	Avg TW	Avg solved TW			
		S1	S2	S3	S4
auction paths	39	39	39	39	39
auction regions	109	109	109	109	109
auction scheduling	98	95	95	95	95
pseudo factor	38	38	38	38	38
pseudo miplib	186	13	13	13	13
qcp	362	N/A	N/A	351	364
planning	136	136	136	136	136
spot5 dir	77	16	16	16	16
spot5 log	79	16	16	16	16

Table 3: Treewidth information of weighted partial Max-SAT problems solved by Clone. All treewidths reported are upper bounds of actual treewidths.

Family	Avg TW	Avg solved TW			
		S1	S2	S3	S4
maxclique random	126	131	132	132	132
maxclique structured	329	132	189	180	121
maxone 3sat	84	87	86	86	86
maxone structured	69	51	51	59	58
pseudo garden	59	11	11	11	11
pseudo logic-synthesis	566	N/A	255	255	255
pseudo primes-dimacs-cnf	141	101	100	101	100
pseudo routing	166	105	105	105	138
maxcsp dense-loose	62	49	49	49	49
maxcsp dense-tight	30	30	30	30	30
maxcsp sparse-loose	45	42	43	43	44
maxcsp sparse-tight	27	27	27	27	27
wqueens	131	68	68	86	86

Table 4: Treewidth information of partial Max-SAT problems solved by Clone. All treewidths reported are upper bounds of actual treewidths.

the treewidths of the problems. Nevertheless, for most problem families, similar treewidth pattern can be found in problems solved by MiniMaxSAT. That is, the average treewidths of solved problems are smaller than the average treewidths of the family.[4] Nevertheless, when we took a close look at the performance of MiniMaxSAT on some problems, we found evidence to the contrary. For example, consider Table 5, in which the running time (in seconds) of each solver on selected problems are shown, along with the computed treewidth of each problem. All versions of Clone were able to solve these problems relatively easily,

4 In fact, MiniMaxSAT solved many families completely. Hence, the average treewidth of solved problems is the same as the family's average treewidth on each of these families.

Problem	Computed TW	Running time (s)				
		S1	S2	S3	S4	S5
29.wcsp.dir	28	0.49	0.82	0.82	0.79	T/O
404.wcsp.dir	23	4.89	5.26	5.77	4.87	T/O
404.wcsp.log	23	24.41	24.49	23.30	18.52	T/O
503.wcsp.dir	12	29.84	30.32	33.65	4.93	T/O
MANN_a27.clq	26	10.07	10.49	11.80	12.04	T/O
normalized-ssa7552-158.opb.msat	16	8.656	7.402	14.09	9.044	T/O

Table 5: Running time of all solvers on some weighted and unweighted partial Max-SAT problems with low treewidths.

because of the low treewidths. However, MiniMaxSAT could not solve any of them within the time-out limit.

Next, in Tables 6 and 7, we report the average problem size and the average percentages of different clause types in each family of the weighted and unweighted partial Max-SAT categories from the evaluation. We categorize clauses into the following categories:

1. Hard clauses. These clauses are mandatory, according to the problem description.
2. Unit soft clauses. These clauses represents weighted constraints imposed on literals.
3. Non-unit soft clauses. These clauses are the remaining clauses in the problem.

Family	Average		Average clause percentage		
	# variables	# clauses	Hard	Unit soft	Non-unit soft
auction paths	121	1,497	90.74	9.26	0.00
auction regions	167	9,409	97.98	2.02	0.00
auction scheduling	160	5,760	97.10	2.90	0.00
pseudo factor	1,083	3,060	99.69	0.31	0.00
pseudo miplib	7,948	31,358	96.03	3.97	0.00
qcp	576	6,575	34.42	0.34	65.22
planning	658	18,442	76.37	2.75	20.88
spot5 dir	924	10,155	91.71	8.29	0.00
spot5 log	553	9,970	91.02	5.04	3.93

Table 6: Average problem size and percentages of clause types in each family of the weighted partial Max-SAT problems from the evaluation.

These results reveal one interesting fact: most problems in these categories are overwhelmed with hard clauses. In 7 out of 9 families in the weighted partial Max-SAT category, more than 90 percent of the clauses are hard on average. The same pattern occurs in 7 out of 13 families of the partial Max-SAT category. This observation partially explains why solvers that heavily employ SAT techniques, such as MiniMaxSAT, and PMS, performed

Family	Average		Average clause percentage		
	# variables	# clauses	Hard	Unit soft	Non-unit soft
maxclique random	150	5,738	95.33	4.67	0.00
maxclique structured	484	50,093	94.23	5.77	0.00
maxone 3sat	150	575	72.80	27.20	0.00
maxone structured	504	2,624	80.87	19.13	0.00
pseudo garden	1,486	2,972	50.00	50.00	0.00
pseudo logic-synthesis	1,948	3,413	49.34	50.66	0.00
pseudo primes-dimacs-cnf	1,252	10,785	73.65	26.36	0.00
pseudo routing	2,502	7,244	92.57	7.43	0.00
maxcsp dense-loose	124	941	37.25	0.00	62.75
maxcsp dense-tight	68	817	23.05	0.00	76.95
maxcsp sparse-loose	159	993	45.90	0.00	54.10
maxcsp sparse-tight	75	778	26.95	0.00	73.05
wqueens	160	2,600	93.30	6.70	0.00

Table 7: Average problem size and percentages of clause types in each family of the partial Max-SAT problems from the Max-SAT evaluation.

well in these categories of the evaluation.[5] Moreover, for many families in both categories, there are many soft unit clauses. The presence of many unit soft clauses should benefit those Max-SAT solvers that based their bound computation heavily on unit propagation, because they give more opportunities for unit propagation to be applied initially (which could substantially contribute to bounds computed later). Examples of such solvers are MiniMaxSAT [2], PMS [7], W-MaxSatz [41], and LB-SAT [3]. Note that unit propagation does not directly affect the bounds computed by Clone. Clone only uses unit propagation as a mechanism for deriving forced assignments (see Section 3.2).

6. Effects of Other Properties on Performance

The results in the previous section guided us to undergo further investigations on how treewidths, clause types, and other aspects of Max-SAT problems affect solvers' performance. In the following experiments, we consider weighted partial Max-SAT problems with relatively small treewidths. One of the main goals of these experiments is to confirm MiniMaxSAT's insensitivity to problem structure. All problems used in the following experiment were converted from the Most Probable Explanation (MPE) query on randomly generated Bayesian networks [27].[6] The method used for generating these networks is described in the Appendix.

In this experiment, we considered the version of Clone with all techniques included (S4) and two other available Max-SAT solvers: MiniMaxSAT, and Toolbar [44], which was one of the best-performing solvers in the 2006 evaluation [12]. Note that both MiniMaxSAT

5 MiniMaxSAT was the best solver in both of these categories, while PMS, which is an unweighted partial Max-SAT solver, was the third-best solver in the unweighted partial Max-SAT category.

6 The problems and generater used in this experiment are available at `http://www.cs.ucla.edu/~thammakn/mpe-maxsat-benchmarks`.

and Toolbar utilize bound computation approaches that are different from the one used by Clone. This experiment was performed on a machine with 2.4 GHz processor and 4GB of RAM. The time-out limit was set to 1800 seconds per problem. We run both MiniMaxSAT and Toolbar using their default settings.

Family	Average			Solved problems (median running time (s))		
	Var	Cls	TW	Clone	MiniMaxSAT	Toolbar
random-net-30-2	83.4	584.3	10.4	10(2.78)	9(59.75)	1(1293.41)
random-net-30-3	83.5	1264.8	13.3	10(214.69)	6(207.04)	0(-)
random-net-30-4	82.6	2516.4	16.7	7(167.40)	10(415.27)	0(-)
random-net-30-5	84.7	4564.9	19.7	4(326.89)	5(165.17)	0(-)
random-net-40-2	112.2	807.8	10.4	10(35.50)	2(420.68)	0(-)
random-net-40-3	111.5	1681.3	13.4	5(434.81)	0(-)	0(-)
random-net-40-4	114.5	4006.5	17.3	0(-)	0(-)	0(-)
random-net-50-2	136.9	955	10.5	10(36.69)	0(-)	0(-)
random-net-50-3	144.9	2437.2	14.2	0(-)	0(-)	0(-)
random-net-60-1	153	528.1	7	10(1.64)	10(2.55)	0(-)
random-net-60-2	168.6	1179.4	10.6	5(44.76)	0(-)	0(-)
random-net-60-3	169.9	2516.7	14	0(-)	0(-)	0(-)
random-net-80-1	205.9	721.2	7	10(1.96)	10(29.01)	0(-)
random-net-80-2	225.1	1577.1	10.9	1(13.12)	0(-)	0(-)
random-net-100-1	247.8	860.7	7	10(2.12)	8(433.13)	0(-)
random-net-100-2	284.6	2049.8	10.9	0(-)	0(-)	0(-)
random-net-120-1	307.3	1076.2	7	10(2.34)	1(14.53)	0(-)
random-net-140-1	364.6	1278.9	7	10(2.70)	1(1478.33)	0(-)
random-net-160-1	420	1464.2	7	10(2.93)	0(-)	0(-)

Table 8: Performance of solvers on MPE problems with low treewidths.

Table 8 shows the performance of the solvers on some representative families. The name of each family has the format "random-net-N-C", where N is the number of variables in the Bayesian network, and C is the maximum number of parents of any node in the network (the higher this value, the higher the treewidth). In this table, the number of problems solved by each solver in each family is shown, along with some information about problems in the family. The second and third columns in this table show the average number of variables and clauses in each family. The fourth column contains the average computed treewidth. Variations in problem sizes and treewidths are very low in each problem family, because they are generated from Bayesian networks with very similar structures. All problems in Table 8 are considered small Max-SAT problems and have very small treewidths. The largest computed treewidth among all problems in this set is 21.[7] Of all 190 problems shown in Table 8, Clone solved 122 problems, while MiniMaxSAT solved only 62 problems, and Toolbar solved only 1 problems.

7 When $C = 1$, the graph of the Bayesian network has treewidth 1. However, due to our conversion method, the treewidth of the constraint graph of the corresponding Max-SAT problem is larger.

This result confirmed our hypothesis that MiniMaxSAT (and Toolbar in this case) is not necessarily sensitive to the treewidth of the problem. For problems with low treewidths, Clone only needs to split a small number of variables, resulting in very small search spaces. Other solvers, on the other hand, were not aware of this aspect of the problems and ended up searching in difficult search spaces.

In this experiment, we also observed that the Max-SAT problems that were converted from the MPE problems tend to have a very large number of distinct soft weights. This is because weights are translated from probabilities in the Bayesian network. Moreover, the majority ($> 75\%$) of clauses are non-unit soft clauses. According to the conversion approach used here, the percentage of non-unit soft clauses increases as the maximum number of parents increases. Table 9 reports the average number of distinct soft weights and average clause percentages of problems in each family. The second column reports the number of problems solved by MiniMaxSAT for the reader's convenience. Based on the results in Table 8, MiniMaxSAT seems to perform well on families "random-net-30-C", which contain relatively small number of variables. However, for other families, MiniMaxSAT only solved problems in those families with relatively small number of distinct soft weights and small percentage of non-unit soft clauses (e.g. "random-net-40-2", "random-net-60-1", "random-net-80-1", and "random-net-100-1"). Even for the "random-net-30-C" groups, the performance of MiniMaxSAT is worse on "random-net-30-5", when the number of distinct soft weights and the percentage of non-unit soft clauses are highest.

At this point, we hypothesized that either the large number of distinct soft weights or the high percentage of non-unit soft clauses are responsible for the poor performance of MiniMaxSAT and Toolbar. In order to test this hypothesis, we conducted the next set of experiments.

6.1 Number of Distinct Soft Weights

The next experiment was designed for studying the effects of the number of distinct soft weights on the performance of Max-SAT solvers. In this experiment, we selected a weighted partial Max-SAT problem from the family "random-net-40-5" and varied the number of distinct soft weights without altering its clause structure. The selected problem has a computed treewidth of 13, > 800 distinct soft weights, and contain 4.83% hard clauses, 1.21% unit soft clauses, and 93.96% non-unit soft clauses.[8] These properties, except the number of distinct soft weights, are kept constant in this experiment. In particular, every hard clause in the problem is kept hard. Then, for a given number of distinct soft weights N, we randomly select N values from the set $\{1, 2, ..., 1500\}$ to be used as soft weights. Then, each soft clause is randomly assigned a weight from the set of selected soft weights with a uniform probability. For each fixed value of N, we generated 50 problems to be used in our experiment, which will focus on how each solver reacts to the varied number of distinct weights. The time-out limit was set to 1200 seconds in this experiment, which was performed on a 3.79 GHz machine with 4GB of memory.

The result of this experiment are shown in Table 10. This result seems to indicate that Clone, MiniMaxSAT, and Toolbar find the problems hardest when the number of distinct soft weights is small. Both solvers tend to solve more problems and have lower running

8 We selected this problem because of its manageable treewidth and large number of soft clauses.

Family	MMS. solved	Average			
		# dsw	% hard	% unit soft	% non-unit soft
random-net-30-2	9	509.9	4.88	2.88	92.22
random-net-30-3	6	1024.3	2.37	1.49	96.15
random-net-30-4	10	1760.8	1.24	0.81	97.97
random-net-30-5	5	2494.2	0.78	0.48	98.75
random-net-40-2	2	683.4	4.71	2.68	92.62
random-net-40-3	0	1290.3	2.39	1.65	95.99
random-net-40-4	0	2399.8	1	0.51	98.5
random-net-50-2	0	797.1	4.83	2.72	92.44
random-net-50-3	0	1729.4	2.01	1.25	96.73
random-net-60-1	10	446.2	9.75	5.83	84.4
random-net-60-2	0	958.9	4.79	2.27	92.95
random-net-60-3	0	1760.9	2.33	1.33	96.34
random-net-80-1	10	597.8	9.49	5.49	85.05
random-net-80-2	0	1221	4.81	2.33	92.86
random-net-100-1	8	692.1	9.69	6.11	84.19
random-net-100-2	0	1495.3	4.68	2.03	93.3
random-net-120-1	1	843.9	9.51	5.27	85.23
random-net-140-1	1	983.7	9.46	4.93	85.62
random-net-160-1	0	1098.8	9.54	4.37	86.07

Table 9: Additional information on MPE problem families. Shown here are (i) the average number of distinct soft weights (ii) the average percentage of hard clauses (iii) the average percentage of unit soft clauses and (iv) the average number of non-unit soft clauses.

# soft weights	Total	Solved problems (median running time (s))		
		Clone	MiniMaxSAT	Toolbar
1	50	50 (88.09)	0 (-)	0 (-)
2	50	50 (47.71)	24 (47.10)	12 (6.94)
10	50	50 (15.95)	50 (21.76)	44 (98.63)
50	50	50 (17.10)	50 (26.32)	44 (164.63)
100	50	50 (16.06)	50 (17.24)	49 (130.19)
200	50	50 (16.28)	50 (18.69)	50 (176.66)
500	50	50 (16.65)	50 (22.80)	50 (186.42)

Table 10: Performance of solvers on problems with varied numbers of distinct soft weight.

time as the number of distinct soft weights increases. Our experiments with several other base problems also resulted in the same trend.

Our explanation for this behavior is the fact that a problem with a large number of distinct weights tends to have only small portion of clauses that are weighted heavily. This allows the JW heuristic (in all of the above solvers) to perform really well, because it takes

clause weights into consideration. If only a small portion of clauses have large weights, JW will tend to select variables appearing in these clauses first. This preference makes subsequent bounds close to the actual optimal costs of the partial assignments, because the remaining weights in the problem are relatively small; there is not much room for the bounds to be inaccurate. This result suggests that the large number of distinct soft weights may not be the reason why MiniMaxSAT and Toolbar performed poorly on problems in Table 8.

Note that the results and analysis presented here are only with respect to specific algorithms and do not contradict the fact that weighted Max-SAT is in a higher complexity class than unweighted Max-SAT [45, 46].

6.2 Percentage of Non-Unit Soft Clauses

The next experiment investigates another potential factor–the percentage of non-unit soft clauses. In this experiment, we used a problem from the weighted partial Max-SAT category of the evaluation as the base problem (WCSP\SPOT5\LOG\29.wcsp.log.wcnf). This problem contains 101 variables and 692 clauses. It also contains many hard clauses with 2-4 literals and no unit hard clause. We then varied the amount of non-unit soft clauses in the problem by randomly selecting a certain number of hard clauses and turning them into soft clauses.[9] The number of distinct soft weights was fixed at 2 in all generated problems, because we wanted the problems to be relatively difficult (based on the results of the previous experiment). The percentage of unit soft clauses is also fixed at 0.09%.[10] In this problem, the size of non-unit soft clauses ranges from 2 to 4. For each amount of non-unit soft clauses, we generated 50 problem instances.

% non-unit soft	Total	Solved instances (median running time (s))		
		Clone	MiniMaxSAT	Toolbar
15	50	50 (2.06)	50 (125.80)	50 (244.88)
20	50	50 (2.30)	50 (162.44)	50 (255.11)
30	50	50 (3.40)	49 (246.89)	50 (349.19)
40	50	50 (5.20)	50 (8.55)	50 (437.61)
60	50	50 (16.93)	50 (37.17)	32 (825.11)
70	50	50 (28.99)	50 (85.78)	14(918.94)
80	50	50 (51.81)	50 (186.27)	3(775.16)
85	50	50 (57.37)	49 (293.58)	3(1008.88)
90	50	50 (104.40)	39 (478.07)	4 (934.59)

Table 11: Performance of Clone, MiniMaxSAT, and Toolbar on problems with varied amount of non-unit soft clauses.

Table 11 reports the results of this experiment. The first column indicates the percentage of non-unit soft clauses in each family. The number of problems solved by each solver,

9 If we picked a base problem with a low percentage of hard clauses, we would need to increase the number of hard clauses instead. In this case, the resulting problems might not contain any solution.

10 We simply kept the unit soft clauses in the original problem. Since there was no unit hard clause in the base problem, our problem generation could not produce more unit soft clauses.

along with the median running time on solved problems, is shown in the last three columns. Let P denotes the percentage of non-unit soft clauses. The performance of Clone and Toolbar clearly decreases as P increases. MiniMaxSAT, on the other hand, has a more irregular behavior. When P is low ($\leq 30\%$), problem hardness seems to increase with P. However, MiniMaxSAT suddenly performs quite well once the value of P reaches 40%. Then, after that point on, MiniMaxSAT performs more poorly as P increases. When $P = 90\%$, Clone solved all problems with median running time of 104 seconds, MiniMaxSAT solved 39 problems with median running time 478 seconds, and Toolbar solved 4 problems with median running time of 935 seconds. In any case, these results seem to be consistent with the behaviors of MiniMaxSAT and Toolbar reported in Table 8. All problems used in that experiment have high percentage of non-unit soft clauses (70-90%), resulting in relatively poor performance of both solvers.

We believe the reason that Clone performs relatively well when P is low is because the bounds produced by Clone in these problems tend to be tight. The source of inaccuracy in Clone's bound computation comes from the fact that, in a relaxed problem, constraints can be satisfied too easily (by the clone variables, each of which appears in only one clause). This directly results in bound inaccuracy whenever such constraints are soft. When these constraints are hard, however, this situation should not have as much impact on bound inaccuracy. This is because every hard constraint has to be satisfied in any solution. However, it may cause variables mentioned in hard constraints to appear more flexible than they actually are. This may or may not affect the values of the bound computed. Therefore, when most clauses in a Max-SAT problem are hard, we may expect Clone to perform relatively better. Many SAT techniques used in Clone, such as unit propagation, should also become most effective whenever P is low, as there are more hard clauses.

Toolbar utilizes local consistency algorithms to make costs more explicit. These explicit costs eventually become bounds used during the search. Most local consistency algorithms used are applicable only to very short clauses. The presence of hard clauses (which have large weights) should have a positive impact on Toolbar's performance, even though it does not directly utilize unit propagation. Hard clauses increase the values of the bounds significantly when violated and allow the solver to prune more often, thus reducing the size of the search space that Toolbar needs to explore. The performance of Toolbar in this experiment reflects the fact that its bound computation method is most effective when there are fewer non-unit soft clauses.

In the case of MiniMaxSAT, the solver's behavior is less straightforward to analyze. Again, let P denotes the percentage of non-unit soft clauses in a problem. Intuitively, it should be the case that as the number of non-unit soft clauses increases, it should be more difficult for MiniMaxSAT to derive tight bounds. This is because problems with high P reduce MiniMaxSAT's chance of applying unit propagation in its bound computation (see [2]). More assignments are then needed before unit propagation can be applied, which means that the solver will spend more time deeper in the search tree. However, this does not explain the sudden increase in performance when $P = 40\%$. We hypothesize that, for this particular base problem, when P reaches 40%, the underlying SAT problem (if we consider only the hard clauses) suddenly becomes easy. We noticed that the number of conflicts reported by MiniMaxSAT dropped sharply (approximately by an order of magnitude) when the value of P went from 30% to 40%. In our additional experiments with different base

problems, we found that all solvers exhibited the same trend (i.e. problems with more non-unit soft clauses appeared harder) and that the performance of MiniMaxSAT degraded consistently as the percentage of non-unit soft clauses increased.

This above analysis of MiniMaxSAT also explains why, in Table 8, MiniMaxSAT tends to perform well on networks in which each node as a small number of parents. For such problems, most clauses are either unit or binary, allowing the lower bound computation in MiniMaxSAT to be effective.

6.3 Discussions

In this section, we have shown some key results that allow us to better understand how various properties of weighted Max-SAT problems can be affect their difficulty. First, we showed that other state-of-the-art Max-SAT solvers (MiniMaxSAT and Toolbar) are not necessarily sensitive to the treewidths of the problems. We gave examples of problem families with very low treewidths that neither solvers could solve. Second, we showed empirically how the number of distinct soft weights affect problem hardness as perceived by the considered Max-SAT solvers. Lastly, our study on how the percentage of non-unit soft clauses affect solvers' performance revealed that a Max-SAT problem could be made harder (for the solvers considered) by increasing the percentage of non-unit soft clauses. We believe that most of the analysis presented here should generalize to other state-of-the-art solvers that employ similar core techniques as well.

All these three properties (treewidth, number of distinct soft weights, and % non-unit soft clauses) of Max-SAT problems could have significant impacts on the difficulty of problems. Hence, they could provide new perspectives for evaluating the difficulty and diversity of weighted Max-SAT problems.

7. Conclusions

In this paper, we described our weighted Max-SAT solver, Clone, which participated in the Max-SAT evaluation 2007. We performed careful analysis on the performance of our solvers in the presence of new techniques on the problems used in the evaluation. Our analysis led us to investigate how certain properties of Max-SAT problems affect solvers' performance. Our study has shown that, unlike Clone, other state-of-the-art solvers do not take advantage of treewidth information. Moreover, we have identified two other properties that could be used to adjust the difficulty level of weighted Max-SAT problems: number of distinct soft weights and percentage of non-unit soft clauses.

A. Bayesian Networks and MPE

In this section, we give a formal definition of Bayesian networks and the most probable explanation query, which was referred to in Section 6. The following definitions are taken from [47].

Definition A.1. An instantiation of a set of variables is a function that assigns a value to each variable in the set. Two instantiations are compatible if they agree on the assignment of all the variables that they have in common.

Definition A.2. A conditional probability table (CPT) T for a variable V with a set of parent variables $\mathbf{P}$ is a function that maps each instantiation of $V \cup \mathbf{P}$ to a real value in [0,1] such that for any instantiation $\mathbf{p}$ of $\mathbf{P}$, $\sum_v T(\{v\} \cup \mathbf{p}) = 1$ where v ranges over the values of V.

Definition A.3. A Bayesian network is a pair $(\mathcal{G}, \mathcal{P})$ where $\mathcal{G}$ is a directed acyclic graph whose nodes are variables and $\mathcal{P}$ is a set which contains the CPT of each variable in $\mathcal{G}$, where the parents of each CPT correspond to the parents of the corresponding variable in the $\mathcal{G}$.

Note that, in contrast to a propositional variable, a Bayesian network variable does not need to be binary. Given a Bayesian network and an instantiation of a subset of the network variables (the evidence), the *most probable explanation* (MPE) query asks for the (not necessarily unique) complete variable instantiation with the highest probability that is compatible with the evidence. We used the approach proposed in [47] to convert each MPE problem into a weighted partial Max-SAT problem. According to this approach, for each of the N variables V_i in the Bayesian network (network variables) with domain $D = \{d_1, d_2, ..., d_n\}$, we have Boolean variables v_{ij}, for $1 \leq i \leq N$,$1 \leq j \leq n$. v_{ij} stands for whether V_i is assigned the value d_j. We then need hard clauses to ensure that every network variable is assigned exactly one value from its domain in any complete instantiation. The evidence (a partial assignment) is asserted as another set of hard clauses. Probabilities from the Bayesian network are translated into clause weights. In this approach, the optimal weight in the resulting Max-SAT problem corresponds to the negative logarithm of the maximum probability of any complete assignment.

B. Generation of Random Bayesian Networks

In this section, we describe the method that we used for generating Bayesian networks for the experiments in Section 6. As described earlier, each Bayesian network consists of two major components: the network graph (DAG) and the conditional probability tables (CPTs). The inputs of this process are the target number of network variables and the maximum number of parents of each node. The network is generated by first creating a node for each of network variables and ordering them arbitrarily. Each node is then assigned a random number of parents between 1 and the supplied maximum number of parents with uniform probability. Once the number of parents of every node is determined, each node's parents are randomly chosen. To ensure that the graph remains acyclic, the parent's of a node are chosen randomly only from the set of nodes which precede that given node in the ordering. After that, each network variable is given a random domain size (the number of possible values the variable can take), which ranges uniformly between 2-4. Finally, the CPT of each node is initialized with random probabilities and then normalized appropriately.

References

[1] Pipatsrisawat, K., Darwiche, A.: Clone: Solving weighted max-sat in a reduced search space. In: Proceedings of Twentieth Australian Joint Conference on Artificial Intelligence. (2007)

[2] Viaga, F.H., Larrosa, J., Oliveras, A.: Minimaxsat: a new weighted max-sat solver. In: Proceedings of SAT'07. (2007)

[3] Lin, H., Su, K.: Exploiting inference rules to compute lower bounds for max-sat solving. In: IJCAI. (2007) 2334–2339

[4] Larrosa, J., Heras, F., de Givry, S.: A Logical Approach to Efficient Max-SAT solving. ArXiv Computer Science e-prints (November 2006)

[5] Ramírez, M., Geffner, H.: Structural relaxations by variable renaming and their compilation for solving mincostsat. In: Proceedings of 13th International Conference on Principles and Practice of Constraint Programming (CP-07).

[6] Alsinet, T., Manya, F., Planes, J.: A max-sat solver with lazy data structures. In: Proc. of 9th Ibero-American Conf. on Artificial Intelligence. (2004)

[7] Argelich, J., Manya., F.: Partial max-sat solvers with clause learning. In: Proceedings of SAT'07. (2007) 28–40

[8] Miyazaki, S., Iwama, K., Kambayashi, Y.: Database queries as combinatorial optimization problems. In: CODAS. (1996) 477–483

[9] Li, X.Y.: Optimization algorithms for the minimum-cost satisfiability problem. PhD thesis (2004) Chair-Matthias F. Stallmann and Chair-Franc Brglez.

[10] Cha, B., Iwama, K., Kambayashi, Y., Miyazaki, S.: Local search algorithms for partial MAXSAT. In: AAAI/IAAI. (1997) 263–268

[11] Choi, A., Zaitlen, N., Hahn, B., Pipatsrisawat, K., Darwiche, A., Eskin, E.: Efficient genome wide tagging by reduction to sat. Paper under review.

[12] Argelich, J., Li, C.M., Manya, F., Planes, J.: First evaluation of max-sat solvers http://www.iiia.csic.es/∼maxsat06/.

[13] Argelich, J., Li, C.M., Manya, F., Planes, J.: Second evaluation of max-sat solvers http://www.maxsat07.udl.es/.

[14] Fu, Z., Malik, S.: On solving the partial max-sat problem. In: Proc. of SAT'06

[15] Le Berre, D.: Sat4j project homepage http://www.sat4j.org/.

[16] de Givry, S., Heras, F., Zytnicki, M., Larrosa, J.: Existential arc consistency: Getting closer to full arc consistency in weighted csps. In: IJCAI. (2005) 84–89

[17] Li, C.M., Manya, F., Planes, J.: New inference rules for max-sat. JAIR (2007)

[18] Li, C.M., Manya, F., Planes, J.: Exploiting unit propagation to compute lower bounds in branch anb bound max-sat solvers. In: Proceedings of CP'2005. (2005)

[19] Larrosa, J., Heras, F.: Resolution in max-sat and its relation to local consistency in weighted csps. In: Proc. of the Intnl. Jnt. Conf. on Artfcl. Intel. (2005) 193–198

[20] Darwiche, A., Marquis, P.: A knowlege compilation map. Journal of Artificial Intelligence Research **17** (2002) 229–264

[21] Darwiche, A.: New advances in compiling CNF to decomposable negational normal form. In: Proceedings of European Conference on Artificial Intelligence. (2004)

[22] Darwiche, A.: Decomposable negation normal form. Journal of the ACM **48**(4) (2001) 608–647

[23] Darwiche, A., Marquis, P.: Compiling propositional weighted bases. Artificial Intelligence **157**(1-2) (2004) 81–113

[24] Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers, Inc., San Mateo, California (2003)

[25] Robertson, N., Seymour., P.D.: Graph minors ii: Algorithmic aspects of treewidth. J. Algorithms **7** (1986) 309–322

[26] Choi, A., Chavira, M., Darwiche, A.: Node splitting: A scheme for generating upper bounds in bayesian networks. In: Proceedings of UAI'07. (2007)

[27] Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers, Inc., San Mateo, California (1988)

[28] Suermondt, H.J., Cooper, G.F.: Probabilistic inference in multiply connected networks using loop cutsets. International Journal of Approximate Reasoning **4** (1990) 283–306

[29] Darwiche, A.: The c2d compiler. Available at http://reasoning.cs.ucla.edu/c2d/.

[30] Mackworth, A.: Consistency in networks of relations. Artificial Intelligence **8**(1) (1977) 99–118

[31] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: 39th Design Automation Conference (DAC). (2001)

[32] Stallman, R., Sussman, G.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intel. **9** (1977)

[33] Marques-Silva, J., Sakallah, K.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers (5) (1999) 506–521

[34] Bayardo, R.J.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of AAAI'97. (1997) 203–208

[35] Ryan, L.: Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University (2004)

[36] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD. (2001) 279–285

[37] McMillan, K.L.: Applying sat methods in unbounded symbolic model checking. In: CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (2002) 250–264

[38] Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT. (2003) 502–518

[39] Selman, B., Kautz, H.: Walksat home page. http://www.cs.rochester.edu/u/kautz/walksat/.

[40] Wang, J.: A branching heuristic for testing propositional satisfiability. In: Systems, Man and Cybernetics, 1995. 'Intelligent Systems for the 21st Century'., IEEE International Conference on. (1995) 4236–4238

[41] Argelich, J., Li, C.M., Manya, F., Planes, J.: Poster for the second evaluation of max-sat solvers http://www.maxsat07.udl.es/ms07.pdf.

[42] Rose, D.: A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. Graph Theory and Computing (1972) 183–217

[43] Kjaerulff, U.: Triangulation of graphs - algorithms giving small total state space. Technical report, Department of Mathematics and Computer Science, Strandvejan, DK 9000 Aalborg, Denmark (1990)

[44] Heras, F., Larrosa, J., de Givry, S., Schiex, T.: Toolbar max-sat solver homepage http://mulcyber.toulouse.inra.fr/projects/toolbar/.

[45] Krentel, M.W.: The complexity of optimization problems. In: STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing, New York, NY, USA, ACM (1986) 69–76

[46] Papadimitriou, C.H.: Computational Complexity. Addison Wesley (1994)

[47] Park, J.D.: Using weighted max-sat engines to solve mpe. In: Proceedings of the 18th National Conference on Artificial Intelligence (AAAI). 682–687