Solving the Minimum-Cost Satisfiability Problem Using SAT Based Branch-and-Bound Search

Zhaohui Fu* Sharad Malik
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544, USA
{zfu,sharad}@Princeton.EDU

ABSTRACT

Boolean Satisfiability (SAT) has seen many successful applications in various fields, such as Electronic Design Automation (EDA) and Artificial Intelligence (AI). However, in some cases it may be required/preferable to use variations of the general SAT problem. In this paper we consider one important variation, the Minimum-Cost Satisfiability Problem (MinCostSAT). MinCostSAT is a SAT problem which minimizes the cost of the satisfying assignment. MinCostSAT has various applications, e.g. Automatic Test Pattern Generation (ATPG), FPGA Routing, AI Planning, etc. This problem has been tackled before – first by covering algorithms, e.g. *scherzo* [3], and more recently by SAT based algorithms, e.g. *bsolo* [16]. However the SAT algorithms they are based on are not the current generation of highly efficient solvers. The solvers in this generation, e.g. Chaff [20], MiniSat [5] etc., incorporate several new advances, e.g. two literal watching based Boolean Constraint Propagation, that have delivered order of magnitude speedups. We first point out the challenges in using this class of solvers for the MinCostSAT problem and then present techniques to overcome these challenges. The resulting solver MinCostChaff shows order of magnitude improvement over several current best known branch-and-bound solvers for a large class of problems, ranging from Minimum Test Pattern Generation, Bounded Model Checking in EDA to Graph Coloring and Planning in AI.

Categories and Subject Descriptors:

B.6.3 **[Design Aids]**: Optimization, Verification

General Terms:

Algorithms, Optimization, Verification

Keywords:

Boolean Satisfiability, MinCostSAT, Branch-and-Bound

1. INTRODUCTION

In the last decade Boolean Satisfiability (SAT) has seen many great advances, including non-chronological backtracking, conflict driven clause learning and efficient Boolean Constraint Propagation (BCP). As a consequence many applications have been able to successfully use SAT as a decision procedure to determine if a specific instance is satisfiable or unsatisfiable. However, there are many other variations of the SAT problem that go beyond this decision procedure. For example, the Minimum Size Test Pattern Generation Problem (MinSTPG) [6] looks for the test pattern with a *minimum number* of specified primary input assignments for detecting a specific stuck-at fault. This paper examines a generalization of this problem named the Minimum-Cost Satisfiability Problem (MinCostSAT) [13].

In the past decade, there have been various solvers, for example *scherzo* [3], *bsolo* [16], *eclipse* [13] etc., targeting the MinCostSAT problem and many effective techniques have been proposed, e.g. Maximum Independent Set [2,3] or Linear Programming [13,14] based lower bounding, SAT based pruning [16] etc. However, none of these have exploited the latest advances in SAT solvers, particularly two literal watching based fast Boolean Constraint Propagation (BCP), which have enabled order of magnitude speedup in SAT solvers. One major reason is that two literal watching based BCP is fundamentally different from any of the previous methods, e.g. counter based BCP. Many operations, e.g. the literal counter updating of clause, have been eliminated for performance reasons in two literal watching based SAT solvers. However, these data structures are essential for all previously successful MinCostSAT solvers, e.g. *scherzo* [3], *bsolo* [16], *eclipse* [13] etc. For example, most of these solvers use branch-and-bound search, which usually uses the set of unresolved (currently unsatisfied) clauses to compute the lower bound. The set of unresolved clauses can be quickly identified by checking the literal counters. Without the literal counters, the lower bound computation of a branch-and-bound search becomes the greatest challenge faced by any two literal watching based SAT solver, which does not maintain the satisfied/unsatisfied state of a clause and thus cannot tell whether a clause is satisfied or not without the computationally expensive operation of checking all its literals. This paper examines these challenges in detail and proposes various methods to solve them efficiently.

1.1 Our Contribution

We present MinCostChaff, a SAT based branch-and-bound MinCostSAT solver based on a contemporary SAT solver, zChaff. MinCostChaff implements many novel ideas that are specially tailored for zChaff (and other solvers of this generation):

1. Incorporating advanced techniques in SAT, e.g. two literal watching based BCP [20], variable branching selection based on VSIDS [20] etc., with branch-and-bound search.
2. A *static* Maximum-Independent Set (MIS) (as compared to a dynamic MIS used by previous solvers) for the lower bounding function. This is only pre-computed once before the search starts and is dynamically maintained during the search. We show that the static MIS is valuable in exploiting the two literal watching based BCP.

*Z. Fu is on leave from the Department of Computer Science, National University of Singapore, Singapore 117543.

ICCAD'06, November 5-9, 2006, San Jose, CA

3. An adaptive lower bounding scheme that dynamically adjusts itself to suit the different instances of the MinCostSAT problem.

Extensive experiments using different types of benchmarks show that MinCostChaff outperforms the best known branch-and-bound based solvers with *order of magnitude* improvements on a large class of problems.

2. BACKGROUND REVIEW

We define the MinCostSAT problem as follows:

DEFINITION 1. *A MinCostSAT problem instance P is: Given a Boolean formula* φ *with n variables* $x_1, x_2, \ldots, x_n$ *with cost* $c_i \geq 0$*, find a variable assignment* $X \in \{0,1\}^n$ *such that X satisfies* φ *and minimizes*

$$\mathcal{C} = \sum_{i=1}^{n} c_i x_i$$

where $x_i \in \{0,1\}$ *and* $1 \leq i \leq n$.

Note that the above definition requires *non-negative cost* for variables with 1 assignment and *zero* cost for variables with 0 assignment, i.e. $c_i^1 \geq 0$ and $c_i^0 = 0$. This, however, does not cause any loss of generality. Assume we have $c_i^1 \geq c_i^0$ (or otherwise we could have an equivalent problem $P^\star$ with $x_i^\star = x_i', c_i^{0\star} = c_i^1, c_i^{1\star} = c_i^0$ and $\mathcal{C}^\star = \mathcal{C}$, i.e. swap x_i with x_i' and c_i^0 with c_i^1). We simplify the problem to P^* such that $x_i^* = x_i, c_i^{1*} = c_i^1 - c_i^0, c_i^{0*} = c_i^0 - c_i^0 = 0$ and $\mathcal{C}^* = \mathcal{C} - \sum_{i=1}^{n} c_i^0$. Finally we have $P^* \equiv P$ with $c_i^{1*} \geq 0$ and $c_i^{0*} = 0$.

The decision version of MinCostSAT is in NP and is NP-complete as the SAT problem is a special case of MinCostSAT with $c_i = 0$ for all i. Bounds of approximability are given by Khanna *et al.* [12] for a special case of MinCostSAT whose $c_i = 1$ for all i.

The difference between MinCostSAT and 0-1 Integer Linear Programming (ILP), which is also known as Linear Pseudo-Boolean Optimization (PBO), is that 0-1 ILP allows coefficients in the constraints (clauses) in addition to the objective function.

MinCostSAT is the generalization of many closely related combinatorial optimization problems that were previously investigated in isolation. For example, the well known Min-One/Max-One Problem [12] is a special case of MinCostSAT with $c_i = 1/-1$ for all i; the Binate Covering Problem [3] is identical to our MinCostSAT problem. There are also problems that can be easily reformulated as the MinCostSAT problem. Examples include Partial MAX-SAT (PMSat) [7,19] and MAX-SAT [11], which can be reformulated with the introduction of slack variables [13].

We will assume that the Boolean formula φ is provided in Conjunctive Normal Form (CNF) as it is the case with most modern SAT solvers

2.1 Classic Covering Algorithms

The earliest important forms of MinCostSAT are the Unate and Binate Covering Problems (UCP/BiCP[1]). BiCP is a synonym of MinCostSAT. UCP has an additional property of every variable x_i only appearing exclusively in one phase, i.e. complemented or uncomplemented, in the problem instance. UCP/BiCP solvers, like *scherzo* [3], use a matrix representation of the problem instance. In UCP you need to *cover* all rows of the matrix (clauses in MinCostSAT) using a minimum number of columns (true variables in MinCostSAT). These methods are typically known as *covering* algorithms, which were once the dominating algorithms for MinCostSAT.

[1]To differentiate from Boolean Constraint Propagation (BCP), we use BiCP to denote the Binate Covering Problem.

scherzo [3] is the most well known branch-and-bound solver in this category. It incorporates many state-of-the-art techniques, including a Maximum Independent Set (MIS) based lower bounding function[2] (which will be discussed it in Section 2.1.2), branch variable selection and various rules for search pruning.

2.1.1 Branch-and-Bound Search

The branch-and-bound search prunes the search space by using a lower bounding function, while an explicitly complete search explores every leaf of the search tree and therefore results in exponential time complexity.

The branch-and-bound search compares the sum of the already incurred cost at current search node ($Cost_{current}$) and the minimum possible cost (*LowerBound*) from the nodes beneath the current one with the best known solution ($UpperBound$) to determine whether to continue the search in the current direction. The classic branch-and-bound procedure is given in Algorithm 1.

Algorithm 1 A Typical Branch-and-Bound Algorithm

```
1:  UpperBound := ∞
2:  while There exists a solution do
3:    if Cost_current + LowerBound ≥ UpperBound then
4:      Backtrack to certain previous decision
5:    end if
6:    if A complete solution is found then
7:      UpperBound := Cost_current
8:    end if
9:    Make a decision d
10:   if d causes any conflict then
11:     Resolve the conflict by backtracking to certain previous
        decision
12:   end if
13: end while
14: Return UpperBound
```

2.1.2 Maximum Independent Set (MIS) Based Lower Bounding Functions

One of the most important components in a branch-and-bound search is the lower bounding function. In order for a branch-and-bound solver to be *complete* (or exact), it is crucial to have a *strict* lower bounding function that never overestimates the lower bound, i.e. the estimated lower bound should never be greater than the actual lower bound. Further, since the lower bounding functions are executed very frequently, the efficiency of computing these is a key performance factor for contemporary MinCostSAT solvers.

An MIS based lower bounding function [2] identifies the MIS of *uncovered* rows of the matrix. The size of the MIS is used as lower bound since one column can at most cover one row in the MIS.

2.1.3 Non-MIS Based Lower Bounding Functions

Coudert proposes a non-MIS based algorithm [3] that guarantees a log-approximation ratio with respect to the minimum solution to the UCP. However, the applicability of Coudert's approximation algorithm is limited for the general MinCostSAT problem, i.e. for the binate case when every variable x_i may appear in both positive and negative phases.

Liao and Devadas propose a Linear Programming Relaxation (LPR) [14] based lower bounding function. Recall that the MinCostSAT is a special case of 0-1 ILP with all coefficients in the

[2]The use of MIS in computing lower bound is often credited to *scherzo*. However, the Unate Covering solver included in the famous Espresso minimizer used it as early as in 1984.

constraints to be 1. If we further relax the constraint on integer solutions, MinCostSAT becomes an LP problem, which can be solved by a general purpose LP solver. The LPR based lower bounding function works by constructing an LP instance using current unresolved clauses. It then solves the instance using a general purpose LP solver like *cplex* [10]. The optimal value of the objective function returned by the LP solver could then be used as the lower bound.

The lower bound returned by the LPR approach is usually more accurate than the MIS based lower bounds. However, since the LPR based lower bounding only guarantees the optimal non-integer solution, which may be arbitrarily far away from the optimal integer solution. Li uses the Cutting Plane (CP) [13] technique to further improve the quality of the LPR lower bound. The CP technique continuously moves the non-integer solution closer to the real optimal integer solution by iteratively applying the Gomory Cut [8].

Both LPR and CP based methods depend on contemporary LP solvers like *cplex* to give better lower bounds than the simple and greedy MIS based method. However, these approaches suffer significant overhead caused by both the construction of the LP instance and the LP solver itself. The trade-off between speed and quality of the lower bound usually has to be determined on a case by case basis (as discussed in Section 4.1).

2.2 SAT Based Algorithms

scherzo is very successful on problems with *hundreds of variables* (columns). However, since the late 90s, there have been many breakthrough techniques, particularly conflict driven clause learning [18] and non-chronological backtracking [18], that have resulted in order of magnitude improvement for SAT solvers. Consequently, SAT has been applied to a range of difficult problems.

2.2.1 SAT Based Branch-and-Bound Search

Overall SAT has been much better studied than covering algorithms. Many ideas used in the classic covering algorithms can be more efficiently implemented in a SAT framework. *bsolo* [16] by Manquinho and Marques-Silva is the first state-of-the-art MinCostSAT solver based on a SAT solver, GRASP [18], and it is capable of handling problems with *thousands of variables*. Further, experiments show that *bsolo* performs very well on *heavily constrained problems*, like the MinSTPG Problem [6]. *bsolo* also uses branch-and-bound search, but with many search pruning techniques specially tailored for a SAT solver.

bsolo uses an MIS based lower bounding function (Algorithm 2) implemented in the GRASP SAT solver. Recall that this lower bounding function relies on the MIS of currently unresolved clauses. GRASP uses a counter based implementation of BCP. The zero (one) counter records the number of false (true) literals in a clause. When a variable is assigned or unassigned, counters from corresponding clauses are updated. Therefore it is easy to tell whether a clause is satisfied or not by checking its true literal counter, without scanning any literals in the clause.

2.2.2 MIS Based Lower Bounding Functions

The MIS lower bounding functions used here are similar to the ones used in covering algorithms except the MIS in a SAT based algorithms is a set of *independent unresolved unate* clauses. An *unresolved* clause has unassigned variables and is currently unsatisfied. The term *independent* means that there is no sharing of any Boolean variables between any two clauses in the MIS. For example, clauses $(x_1 + x_2 + x_3)(x_4 + x_5)$ form an MIS and they are satisfied iff at least two variables (one from each clause as they are disjoint or *independent*) are assigned to true. The MIS is usually constructed using a greedy heuristic as shown in Algorithm 2.

Algorithm 2 Construction of the MIS (φ)

```
1: MIS := ∅
2: φ := φ \ {satisfied clauses}
3: while φ ≠ ∅ do
4:    ω := HeuristicChooseClause(φ)
5:    MIS := MIS ∪ ω
6:    φ := φ \ {clauses sharing variables with ω}
7: end while
8: Return MIS
```

Given an MIS of *unate* clauses, i.e. clauses with only positive literals, the lower bound is given by

$$Cost(MIS) = \sum_{\omega \in MIS} Weight(\omega) \quad (1)$$

where ω is a clause in the MIS and

$$Weight(\omega) = \min_{x_i \in \omega} c_i \quad (2)$$

Algorithm 2 finds the MIS by repeatedly selecting a clause and removing all the intersecting clauses from problem instance φ until no more clauses are left. *bsolo* uses a greedy heuristic that always chooses the clause (Line 4 of Algorithm 2) that maximizes the following ratio:

$$CostRatio(\omega) = \frac{Weight(\omega)}{|\{x_i \mid x_i \in \omega\}|} \quad (3)$$

The rationale behind this heuristic is that smaller clauses tend to intersect with fewer other clauses and the resulting MIS is likely to have more clauses and hence provides a larger lower bound.

2.2.3 Encoding Based MinCostSAT Solvers

In an alternate approach, a SAT based linear search [1,15] translates the MinCostSAT instance into a SAT instance by using an auxiliary counter to encode the objective function. The counter maps the numerical output into a Boolean variable such that it is true if and only if the numerical value is less than or equal to a certain threshold. This approach then uses linear search starting form the highest possible value of the objective function and iteratively reduces the value of the known cost function until the problem instance becomes unsatisfiable, at which time the minimum value of the objective function is found. However, this approach suffers significant overhead caused by the introduction of the auxiliary counter and it is not competitive on large problems [16].

Better encoding schemes are proposed by *MiniSat+* [5], which is a new PBO solver implemented on top of *MiniSat* [5]. It converts the PB constraints into SAT clauses using one of the three methods:

1. Converting the PB constraint to SAT clauses using Binary Decision Diagrams (BDDs).
2. Converting the PB constraint to a network of adders. The sum of the left-hand side of the PB constraint is compared to the right hand side in binary format. Then this circuit is translated into SAT clauses.
3. Converting the PB constraint to a network of sorters and again the circuit for the comparison (in unary format) is constructed and translated to SAT clauses.

MiniSat+ shows impressive results on PB'05 [17]. However, as we will show in Section 4, the efficiency of the translation of PB constraints to SAT clauses using any of the above three methods still depends on the number of non-zero cost variables in the objective function. Therefore such an encoding based method is not efficient for problem instances with a large number of non-zero cost variables.

3. ENGINEERING AN EFFICIENT BRANCH-AND-BOUND MINCOSTSAT SOLVER

SAT has seen many significant advances since the development of GRASP. zChaff [20] first implements the two literal watching based BCP [20] and Variable State Independent Decaying Sum (VSIDS) [20] based decision heuristic. As a result, zChaff is able to handle instances with *tens of thousands of variables*, which is an order of magnitude larger than GRASP.

Compared to the two literal watching, the counter based BCP used in GRASP (and hence *bsolo*) suffers from a significant bookkeeping overhead. The new two literal watching based BCP dramatically reduces this bookkeeping overhead by only updating the clauses where this variable is being *watched*.

The success of zChaff naturally raises the possibility of further improving the performance of current MinCostSAT solvers, like *bsolo*, which is based on GRASP, by incorporating techniques used in zChaff, like the two literal watching based BCP. *However, this poses new challenges which need to be overcome to successfully leverage these advances in SAT. This is precisely the focus of this paper.*

3.1 MIS Based Lower Bounding Function

We choose the MIS based lower bounding function over the LP based (either LPR or CP) functions for two major reasons. First, the MIS based lower bounding function is simple and hence incurs much less overhead than LP based ones. Second, we focus more on large instances that are usually more constrained than general LP instances.

All MIS based lower bounding functions we discussed earlier in Section 2.1.2 use a *dynamic* MIS in the sense that they compute the MIS each time according to the current state of the search, e.g. variable assignments, unresolved clauses etc. The major drawback of the dynamic MIS is the overhead in the repeated construction of the MIS. This is further exacerbated by the two literal watching based BCP. Recall that two literal watching BCP does not maintain the counters for each clause. As a result, it cannot even tell whether a clause is satisfied or not without scanning all literals in this clause. Identifying all unresolved clauses hence requires the scanning of all the literals for all the clauses, which is a very expensive operation as we will show in the experiments in Section 4.

3.1.1 A Pre-computed Static MIS

To overcome the difficulty of computing an MIS dynamically, we propose the use of a static MIS that is pre-computed and dynamically maintained using the current problem state. The rationale is to trade-off little quality of the estimated lower bound for a significant speedup. The *static* MIS still uses the typical MIS construction procedure as presented in Algorithm 2, but it is pre-computed before the branch-and-bound search starts, only clauses from this static MIS are used to compute the lower bound during the search. Therefore it is critical to construct a good static MIS that has an accurate (larger) *expected* lower bound in the future search.

3.1.2 Construction of the Static MIS

Recall that *bsolo* only selects the *unresolved unate* clauses, i.e. clauses with all positive literals left, for the MIS. This constraint usually results in a small yet accurate MIS since each clause contributes to cost of the MIS (i.e. the lower bound). Note that an unresolved unate clause requires only unassigned literals to be positive. However, this constraint of unate clause does not work well for our static MIS because the static MIS is pre-computed before any variable assignment. In other words, an initially non-unate clause may become an unresolved unate clause during the search when all its negative literals are assigned to be false. However, since our MIS is only pre-computed once, a clause that is not in the MIS during its construction will never be in the MIS for the entire search. Hence excluding these initially non-unate clauses could result in a very small static MIS that significantly underestimates the lower bound during the search. We have to relax this constraint of unate clauses during the selection of MIS clauses. For example, clause $(x'_1 + x_2 + x_3 + x_4)$ cannot be in the MIS according to *bsolo* before any variable assignment since x'_1 is a negative literal. However, this clause can be part of the MIS once x_1 is assigned to be true (x'_1 is false). We choose to include this clause in our static MIS, taking into consideration that this clause is likely to become unate during the future search. Note that though we include the negative literals in our MIS clauses, we still enforce the mutually non-intersecting rules among these MIS clauses.

A critical component in the construction of the MIS is the heuristic used to select a clause for the MIS (Line 4 of Algorithm 2). Recall that *bsolo* uses a greedy heuristic that maximizes the ratio (*CostRatio* in Equation 3) of the clause weight (Equation 2) with the number of literals in the clause. (Note that though negative literals are now allowed in the MIS clauses, their cost are always set to 0 due to our previous assumption of $c_i^1 \geq 0$ and $c_i^0 = 0$.) The clause selection heuristic using *CostRatio* favors short clauses as they tend to intersect with fewer other clauses and results in an MIS with more clauses and hence provides a larger lower bound. However, the *CostRatio* only considers the clauses that are useful for computing the immediate lower bound; it does not take into consideration the *expected* lower bound in the future of the search, which is remarkably more important in our static MIS approach. We propose the clause selecting heuristic based on the following measurement:

$$ExpectedCost(\omega) = \frac{\sum_{x_i \in \omega} c_i}{|\{x_i \mid x_i \in \omega\}|} \quad (4)$$

Equation 4 is a better estimation of the cost of the clause in the acutal search because it considers the *expected* cost of a clause in the future. Consider the following simple example with $c_1 = c_2 = c_3 = 100, c_4 = c_5 = c_6 = 1$.

$$\begin{aligned} \omega_1 &= (x_1 + x_2 + x_3 + x_4) \\ \omega_2 &= (x_3 + x_4 + x_5 + x_6) \end{aligned}$$

The heuristic used in *bsolo* considers both clauses ω_1 and ω_2 to be equally preferable since $CostRatio(\omega_1) = CostRatio(\omega_2) = \frac{1}{4}$. This observation is valid only for the current state of the search where both ω_1 and ω_2 contribute 1 to the lower bound. However, in terms of the *expected* cost in the future, clause ω_1, which has more high cost literals, therefore should be more preferable because it is *likely* to give larger cost in the future search (if we assume equal probability for every x_i being true or false). Our measurement in Equation 4 captures this information as $ExpectedCost(\omega_1) = \frac{301}{4} > ExpectedCost(\omega_2) = \frac{103}{4}$. Note that even for cases with $c_i = 1$ for all i, *ExpectedCost* works better because of the presence of negative literals, which always have cost 0, in the clause. A detailed comparison of the lower bounds returned by two different *static* MISs constructed using *CostRatio* and *ExpectedCost* is given in Figure 1, which shows the experiment conducted on the `bmc-ibm-3` benchmark. Details on the benchmark can be found in Table 1 in Section 4. Initially, heuristic using *CostRatio* constructed its MIS with 5796 clauses and *ExpectedCost* has 4365 clauses. The intersection of the curve with the X-axis indicates the discovery of a feasible solution. The better (larger) lower bound is used for each search iteration of the search. Figure 1 shows that the MIS constructed using our new heuristic with *ExpectedCost* is almost always superior than the heuristic *CostRatio* used by *bsolo* by

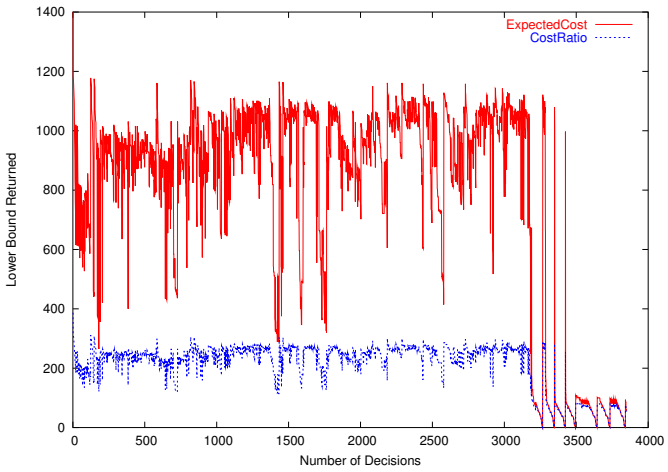

Figure 1: Comparison of lower bounds returned by two different *static* MISs constructed using *CostRatio* (Equation 3) and *ExpectedCost* (Equation 4) respectively.

providing a much larger (and hence more accurate) lower bound, even though the heuristic using *CostRatio* constructs a larger MIS of clauses (5796 vs. 4365).

3.1.3 Computing the Lower Bound using the Static MIS During the Search

After the construction of the static MIS, we show how it is used to compute the lower bound in Algorithm 3. It iterates though all clauses in the MIS and sums up the minimum possible cost to satisfy each clause. Note that Line 6 of Algorithm 3 denotes the two cases where the current clause does not incur any cost. The first case is that the current clause is already satisfied (hence inactive). The second one is that there are unassigned negative literals in this clause and this clause can be satisfied by setting a negative literal to true (corresponding variable setting to false) with no cost.

Algorithm 3 Computing the Lower Bound (*MIS*)

```
1:  LowerBound := 0
2:  for all ω ∈ MIS do
3:    Cost := ∞
4:    for all Literal l ∈ ω do
5:      Let x_i be the variable corresponding to Literal l
6:      if Literal l is TRUE or l is UNASSIGNED and negative
        then
7:        Cost := 0
8:        Break
9:      end if
10:     if Cost > c_i then
11:       Cost := c_i
12:     end if
13:   end for
14:   LowerBound := LowerBound + Cost
15: end for
16: Return LowerBound
```

We do not explicitly update our MIS, e.g. marking satisfied clauses inactive, during the search, but rather check all MIS clauses during the computation of the lower bound. This "lazy" approach works well because a two literal watching based SAT solver does not maintain the list of satisfied clauses for efficiency reasons and one cannot tell what clauses are satisfied after a variable assignment.

3.1.4 Improve the Lower Bounding with Multiple Static MISs

The idea of using one static MIS can be extended to having k different MISs. Multiple static MISs have a higher probability of giving a larger lower bound as compared to a single MIS. Note that k is a small constant number that should be negligible as compared to the total number of decisions. These MISs are still constructed based on our proposed approach of using the *ExpectedCost* criterion, but are different from each other. For instance, the first MIS MIS_1 is constructed in the same way as before, the first clause in the second MIS MIS_2 should be different from the one used in MIS_1 and so on.

3.1.5 No Lower Bounding Function At All?

The pre-computed static MIS eliminates the overhead caused by repeated constructions of MIS. However, profiling shows that as much as 70% of the CPU time is still spent on lower bound computation in certain cases. There are cases where the benefit from lower bounding is overwhelmed by the cost of computation. Naturally we should not use any lower bounding functions for these benchmarks. Examples of these cases are presented in Section 4, which belong to one of the following two types. The first one is the heavily constrained problem with very small solution spaces, i.e. very small number of satisfying assignments. In these cases, the search space is mostly pruned by the existing constraints and makes lower bounding ineffective. The second type includes the problems with many $c_i = 0$, i.e. zero-cost variables. These zero-cost variables make every clause easy to satisfy with low (or zero) cost and the lower bound returned is zero most of the time.

3.1.6 Compute Lower Bound Adaptively

Based on our discussion in the previous section, we propose an adaptive lower bounding scheme by constantly monitoring the values returned by the lower bounding function and disabling itself when the number of consecutive *poor*, e.g. returning 0, lower bounds exceeds a certain threshold, e.g. 100. Note that once the lower bounding function is disabled by the adaptive approach, it will never be enabled again for the rest of the search. This adaptive strategy provides a good balance between the quality of the lower bound and the time used to compute it. Experiments in Section 4 show that this adaptive approach takes the advantages from both persistent lower bounding and no lower bounding.

3.2 Compact Blocking Clause

After an extensive discussion on optimizing the lower bound computation, we focus on the upper bound blocking in this section. The upper bound in a branch-and-bound search corresponds to the best known solution and the upper bounding is implemented using a blocking clause, which prunes the unwanted search space as in *bsolo*. This is also adopted in the same way by MinCostChaff. It is well known that shorter clauses are always preferred by the state-of-the-art SAT solvers as they generate faster and more frequent BCP. We illustrate how to obtain a shorter blocking clause while achieving the same effectiveness with static MIS in MinCostChaff.

The blocking clause consists of variables from the set $\{x'_i | c_i > 0 \text{ and } x_i = 1\}$, i.e. the variable assignments explaining $Cost_{current}$. Manquinho and Marques-Silva [16] propose to include the variables *explaining* the *LowerBound* in the blocking clause as well. The explanation variables are the ones that appear in the set $\{l | l = 0 \text{ and } l \in \omega_j \text{ and } \omega_j \in MIS\}$ as literals. In other words, the literals corresponding to these explaining variables always have false value in the MIS clauses. Recall that only unresolved MIS clauses contribute to the lower bound. If any of these variables is assigned

the opposite value, then some of the MIS clauses will be satisfied and hence the MIS lower bound is no longer valid. Explanation variables capture additional information of the current state of the search and after combining with true non-zero cost variables, they form a complete blocking clause that prunes the unwanted search space. Furthermore, *bsolo* uses other techniques to shorten the blocking clauses by dropping variables while keeping the sum of lower bound and current solution still greater than or equal to the upper bound. This technique is shown to be very effective and is also adopted in MinCostChaff.

3.3 Other Optimization Techniques

Besides the major performance factors discussed above, we are able to further improve the performance of MinCostChaff with the following optimization techniques.

3.3.1 Branch Variable Selection

zChaff uses VSIDS for branching variable selection. VSIDS is a scoring scheme based on the number of occurrences of the variables and picks the unassigned variable with the largest score to branch on. The value of a variable is determined based on its occurrences in positive/negative phases and it is set to the value that satisfies more clauses.

We uses VSIDS to select the branching variable x_i, but the value of x_i is determined according to the VSIDS score only if $c_i = 0$. We assign $x_i = 0$ if $c_i > 0$. This is a greedy method of minimizing the cost, i.e. we always assign the variables with non-zero cost to be false unless we are forced otherwise. This method is consistent with the limit lower bound theorem [4] proposed by Coudert, which says any variable with $c_i \geq Cost_{current} + LowerBound - UpperBound$ is implied to be false.

3.3.2 No Expensive Simplifications

There are several simplification techniques used in classic covering algorithms, e.g. essentiality, row/column domination etc. The essentiality is automatically taken care of by BCP in SAT. *bsolo* considers other simplifications using row/column domination, which are *not* adopted in MinCostChaff. The row/column dominance corresponds to the clause/variable subsumption in SAT. However, clause/variable subsumption is known to be an expensive operation for contemporary SAT solvers and it is very rarely used.

4. EXPERIMENTAL RESULTS

We have conducted extensive experiments on various benchmarks from different categories. The first set of the benchmarks includes:

1. MinSTPG Problem [6]. The problem is to find the test pattern with a *minimum number* of specified primary input assignments for detecting a specific stuck-at fault.
2. Bounded Model Checking (BMC).These are the BMC benchmarks from the SAT benchmark set [9] with the added constraint that $c_i = 1$ for all variables x_i.
3. Graph Coloring with 3 colors [9]. $c_i = 1$ for all variable x_i.
4. AI Planning (Logistics and Blocks World Planning [9]) problems. $c_i = 1$ for all variables x_i.
5. Covering benchmarks. These are classic MCNC [21] BiCP Benchmarks with $c_i = 1$ for all i.

For each of these benchmarks, we present the performance of different variants of MinCostChaff in columns 6-10 as well as the best known solver *MiniSat+*, *scherzo*, *bsolo* and state-of-the-art LP solver *cplex* (used in 0-1 ILP mode) in columns 11-14 of Table 1. Column 6 shows the run time using counter based BCP (same as *bsolo* and GRASP) with dynamic MIS. Columns 7-10 use 2-literal watching based BCP. In particular, column 7 (Dyna.) uses dynamic MIS; column 8 (Static) uses static MIS; column 9 (No LB) does not use any lower bounding function and hence no MIS; column 10 (Adapt.) uses adaptive lower bounding.

All the experiments are conducted on a Dell PowerEdge 700 running Linux Fedora core 1.0 (g++ GCC 3.3.2) with single Pentium 4 2.8GHz, 1MB L2 cache CPU on 800MHz main bus. Time/Memory limits for all solvers is set to be 1hour/1GB. *cplex* version 9.1.0 is used for all experiments.

Table 1 clearly shows that all variants of MinCostChaff solver outperform both *bsolo* and *cplex* on most benchmarks with order of magnitude improvements. MinCostChaff with adaptive lower bounding performs similar to *MiniSat+* on benchmarks with small number of non-zero cost variables. For instance, MinSTPG only introduces non-zero cost variables for the primary inputs and Coloring benchmarks have a small number of total variables (and hence a small number of non-zero cost variables). MinCostChaff performs significantly better than *MiniSat+* on other benchmarks, e.g. BMC and Planning. Counter based BCP shows better results in some MinSTPG benchmarks over two literal watching BCP with *dynamic* MIS. This is mainly due to the fact that the solver with counter based BCP does not need to scan the literals in all clauses, which is very expensive for identifying unresolved clauses. Results from MinSTPG benchmarks show that lower bounding functions are not effective because most variables in these benchmarks have zero cost and the MIS lower bounding function almost always returns zero for lower bound. However, benchmarks from BMC and AI Planning show dramatic performance gain using the lower bounding functions. The solver with adaptive lower bounding takes advantages of both persistent lower bounding and no lower bounding. Solver with multiple MIS does not work well on these benchmarks and the results are not presented due to space limit.

4.1 A Discussion on Covering Benchmarks

Though UCP/BiCP are the earliest forms of MinCostSAT, there are clear distinctions between UCP/BiCP and other MinCostSAT benchmarks we presented in Table 1. For example, all clauses in UCP are unate clauses (only positive literals) and hence all implications are positive, i.e. implying a variable to be true, and there will never be a conflict during the search. In other words, UCP has a large solution space and it is extremely easy to find a satisfying assignment. Though BiCP could include negative literals in the clauses, all the MCNC BiCP benchmarks in Table 1 contain at most one negative literal per clause, which makes them very similar to the UCP benchmarks and thus they have very few conflicts. For example, benchmark `alu4.b` from the covering category has 808 Boolean variables and 1838 clauses (1747 of them have one negative literal) while a typical non-covering benchmark like `misex3_Fb1` has 2348 variables and 12574 clauses.

The strength of the contemporary SAT solvers, e.g. fast BCP, conflict driven clause learning etc., are not effective for this class of loosely constrained problems. Another reason of the poor performance is because our static MIS cannot provide sufficiently accurate lower bounds. Since the solution space is extremely large, an underestimated lower bound could lead to a significant amount of time being wasted on some hopeless search space.

cplex is the best solver in this category of benchmarks and its remarkable performance also shows that the covering benchmarks are more similar to generic ILP instances rather than SAT instances. It is worth mentioning that the current version of *bsolo* has an option to use LPR for lower bounding and it automatically switches to LPR mode based on its own heuristics. For example, it switched to LPR lower bounding during the execution of benchmark `ex5.pi`.

Table 1: Performance comparison of different categories of benchmarks. When solver is time/memory out, the best solution found (if sovler reports it) is presented with *, otherwise -. We noticed (and have confirmed with other researchers) that *scherzo* occasionally gives false optimal solutions (denoted with *) that are 1 larger than the actual optimal solutions. All time is presented in seconds.

Category	Benchmark	Num. Var.	Num. Cls.	Min. Cost	Cnt. BCP	2 Literal Watching Based BCP				Other Solvers			
					Dynamic	Dyna.	Static	No LB	Adapt.	*MiniSat+*	*scherzo*	*bsolo*	*cplex*
Minimum Size Test Pattern Generation	c1908_F469	2826	12735	11	2.59	3.68	0.35	0.31	0.32	0.53	-	32.10	3182.86
	c1908_F953	3540	17796	4	8.01	11.82	0.90	0.65	0.70	0.63	-	32.36	208.58
	c3540_F20	7600	41312	6	48.40	58.19	3.75	4.50	4.79	2.49	-	413.26	6*
	c3540_F45	6000	29076	9	43.35	34.27	5.93	5.75	5.75	2.62	-	3428.20	9*
	c432_F37	966	5054	9	7.32	8.25	1.09	1.04	1.02	0.21	-	29.55	47.95
	c5315_F54	3778	16000	5	21.08	48.29	1.33	0.44	0.61	0.61	-	27.84	4.55
	c6288_F35	6791	30396	4	5.31	4.53	0.27	0.26	0.27	0.97	2751.05	69.96	5*
	c6288_F69	7539	36257	6	6.13	12.97	1.01	0.98	0.98	2.00	-	606.60	10*
	9symml_F6	735	4048	9	0.12	0.14	0.06	0.06	0.06	0.14	-	1.35	74.95
	alu4_Fj	2881	16732	6	0.88	1.29	0.21	0.20	0.21	0.71	95.21*	10.10	2066.50
	alu4_Fl	2908	16980	6	0.90	1.76	0.33	0.31	0.32	0.68	180.37*	8.18	1068.59
	duke2_Fv5	1836	10162	5	0.71	1.71	0.11	0.09	0.09	0.39	-	6.67	157.87
	misex3_Fa0	2051	10088	9	0.61	1.12	0.23	0.22	0.23	0.36	-	7.62	664.52
	misex3_Fb1	2348	12574	8	1.30	2.03	0.46	0.45	0.45	0.48	476.47	13.87	1611.92
	spla_Fv14	2168	10687	8	0.48	0.74	0.10	0.10	0.10	0.36	171.81	7.77	907.18
BMC	2bitcomp-5	125	310	39	25.43	3.36	1.91	19.81	1.92	0.54	22.54	26.53	1.70
	bmc-ibm-2	2810	11683	940	1136.86	155.80	97.22	940*	97.71	210.50	-	20.85	7.51
	bmc-ibm-3	14930	72106	6356	211.45	147.30	1.96	6365*	1.98	6373*	-	76.53	-
Coloring	3col120_5	240	1026	110	24.31	22.02	8.13	8.07	8.13	2.53	-	110*	241.97
	3col140_5	280	1196	124	155.77	53.41	19.97	22.03	19.96	4.46	-	124*	579.73
	3col160_5	320	1366	139	2476.38	241.53	131.90	128.03	132.05	53.98	-	144*	139*
	3col180_5	360	1536	153	157*	153*	2025.98	1997.27	2027.62	153*	-	163*	171*
Planning	logistics.b	843	7301	138	7.26	6.80	0.02	3027.81	0.02	8.85	29.67	138*	87.11
	logistics.c	1141	10719	162	15.73	17.89	0.04	2705.87	0.04	17.86	941.35	162*	162*
	rocket_ext.b	351	2398	69	0.36	0.56	0.02	0.14	0.01	0.36	338.40	5.19	737.26
	bw_large.c	3016	50457	265	14.46	15.67	0.37	0.37	0.38	8.45	-	24.99	165.40
	bw_large.d	6325	131973	431	245.02	141.69	2.92	2.86	2.94	92.10	-	683.77	251.49
Covering	9sym.b	310	976	5	56.16	200.22	583.08	633.16	3205.81	0.68	0.19	62.42	0.11
	alu4.b	808	1838	50	55*	56*	55*	56*	56*	52*	-	50*	189.10
	apex4.a	4317	11912	776	801*	800*	800*	800*	800*	848*	4.07	2358.90	0.33
	ex5.pi	2460	873	65	101*	99*	95*	95*	97*	85*	-	508.90	35.93
	rot.b	1452	2984	115	140*	142*	141*	141*	141*	126*	-	117*	5.81

4.2 Performance on Partial MAT-SAT Benchmarks

We have also conducted experiments on Partial MAX-SAT (PM-SAT) benchmarks [7,19]. PM-SAT has certain constraints (clauses) that are marked as relaxable (soft) and the rest are non-relaxable (hard). The objective is to find a variable assignment that satisfies all non-relaxable clauses together with the maximum number of relaxable ones. We can convert PM-SAT instances to MinCostSAT ones with the introduction of slack variables to the relaxable clauses. The PM-SAT benchmarks we considered include:

1. FPGA Routing. These benchmarks are industrial examples resulting from a SAT based FPGA router. Each unrouted net-arc (single source, single destination) is associated with a unit cost and minimum cost corresponds to the maximum routing of all net-arcs. In these benchmarks $c_i = 1$ only for x_i (slack variable) associated with the routability of the net-arcs.
2. Relaxed UNSAT Instances. We randomly mark some clauses relaxable with a unit cost slack variable, a minimum cost means the problem instance becomes satisfiable with minimum number of clauses left unsatisfied (relaxed).
3. Multiple Property Checking. The outputs of these circuits are *asserted* to be constant 0 or 1 (properties). False assertion incurs a unit cost (by the slack variable) and the minimum cost represents maximum true assumptions.

We include the PM-SAT solver *PMChaff* [7] in our experiment since *scherzo* times out on all these experiments. *PMChaff* first proposes an iterative UNSAT Core elimination approach that iteratively identifies the UNSAT Core and relaxes it using additional relaxation variables. It also uses an encoding based approach (with a tree of adders) for comparison purposes. The best result obtained by either approache is reported in Table 2.

Table 2 again shows that for benchmarks with small number of non-zero cost variables, e.g. FPGA Routing and Multi-Property Checking, MinCostChaff does not perform well since the lower bounding function returns 0 in most cases while some encoding based approaches, e.g. *MiniSat+* or the encoding based approach in *PMChaff*, are more suitable in this case. The relaxed UNSAT benchmarks contain more non-zero cost variables and as a result, encoding based approaches perform poorly due to the large size of the adder/sorter/BDD. The iterative UNSAT Core elimination is still the best approach for this benchmark group.

5. CONCLUSIONS

We have presented an efficient solver MinCostChaff that applies the advanced techniques from contemporary SAT solvers to MinCostSAT using branch-and-bound search. Variants of MinCostChaff with different features were evaluated and MinCostChaff with adaptive lower bounding turns to be the fastest among all these and outperforms *scherzo*, *bsolo* and *cplex* with *order of magnitude* improvements for heavily constrained instances. MinCostChaff does not work well on classic covering benchmarks, where *cplex* shows obvious advantages. The performance of MinCostChaff is comparable to *MiniSat+* on benchmarks with a small number of non-zero cost variables.

Table 2: Performance comparison on some Partial Max-SAT benchmarks. * and - are used in the same way as in Table 1. *scherzo* times out on all these benchmarks. Results NOT from UNSAT Core elimination are marked with * for *PMChaff*.

Category	Benchmark	Num. Var.	Num. Cls.	Min. Cost	Cnt. BCP	2 Literal Watching Based BCP				Other Solvers			
					Dynamic	Dynamic	Static	No LB	Adaptive	*MiniSat+*	*PMchaff*	*bsolo*	*cplex*
FPGA Routing	FPGA_27	3953	13537	3	76.84	149.16	2.99	2.49	2.54	0.88	1.65*	21.29	3*
	FPGA_31	17869	65869	1	-	-	381.20	377.33	376.96	117.39	88.68*	4*	12*
	FPGA_32	2926	9202	3	49.87	92.63	1.92	1.41	1.41	0.78	0.89	6.56	3*
	FPGA_33	9077	32168	3	3369.86	1990.56	50.88	54.44	49.63	2.53	18.65	61.50	4*
	FPGA_39	6352	22865	4	2123.34	423.88	9.89	8.08	8.17	2.21	7.15*	59.07	6*
	FPGA_44	6566	22302	3	6*	1788.41	258.79	254.60	254.53	3.90	8.36*	6*	5*
UNSAT	3pipe	2468	27533	1	692.92	1338.47	44.82	5.80	6.39	131.05	5.12	1*	1*
	4pipe	5237	80213	1	805.33	591.19	35.14	25.23	25.27	141.74	8.45	1*	1*
	5pipe	9471	195452	1	1*	13*	854.85	118.08	126.28	2609.28	18.91	1*	1*
Multi-Property Checking	c2670_1	1567	3409	7	3.97	5.96	0.40	0.19	0.16	0.26	0.05	1.05	106.40
	c5315_1	2609	6816	10	1.88	4.41	0.23	0.13	0.12	0.33	0.09	15.35	208.80
	c6288_1	4723	11700	2	4*	285.63	186.39	187.30	186.28	37.42	81.98*	2*	3*
	c7552_1	4355	10814	5	122.15	111.13	10.43	9.44	9.53	0.76	1.17*	5*	1909.63
	b14_1	10288	28929	1	18.51	34.13	1.09	1.15	0.73	1.16	0.19	4.45	1182.85
	b15_1	9302	26060	2	1155.13	958.44	28.85	9.05	10.13	1.95	0.26	7.55	1308.47
	b20_1	20717	58407	2	15*	3424.49	97.18	60.86	61.56	3.96	0.50	10.88	5*
	c2670_0	1567	3409	7	6.59	11.32	0.77	0.31	0.31	0.23	0.04	1.27	0.10
	c7552_0	4355	10814	6	94.16	249.16	23.08	9.21	21.60	2.38	1.57	2369.50	6*
	b15_0	9302	26060	3	285.69	455.70	12.86	3.60	3.56	2.25	0.22	19.37	260.75

Acknowledgement

We thank Richard Rudell, Olivier Coudert and Vasco Manquinho for providing us various solvers and benchmarks. We also thank several anonymous reviewers for their invaluable suggestions and comments on this paper.

6. REFERENCES

[1] P. Barth. A Davis-Putnam enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max Plank Institute Computer Science, 1995.

[2] O. Coudert. Two-level logic minimization: An overview. *Integration the VLSI journal*, 17(2):97–140, 1994.

[3] O. Coudert. On solving covering problems. In *Proceedings of the 33rd Design Automation Conference (DAC'96)*, pages 197–202, 1996.

[4] O. Coudert and J. C. Madre. New ideas for solving covering problems. In *Proceedings of the 32nd Design Automation Conference (DAC'95)*, pages 641–646, 1995.

[5] N. Eén and N. Sörensson. The MiniSat page, `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`, 2006.

[6] P. Flores, H. Neto, and J. Marques-Silva. An exact solution to the minimum size test pattern problem. *IEEE Transactions on Design Automation of Electronic Systems*, 6(4):629–644, 2001.

[7] Z. Fu and S. Malik. On solving the Partial MAX-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'06), LNCS 4121*, pages 252–265, 2006.

[8] R. Gomory. Outline of an algorithm for integer solution to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[9] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT, `http://www.satlib.org`. *SAT 2000*, pages 283–292, 2000.

[10] ILOG. Cplex homepage, `http://www.ilog.com/products/cplex/`, 2006.

[11] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[12] S. Khanna, M. Sudan, L. Trevisan, and D. Williamson. The approximability of constraint satisfaction problems. *SIAM Journal of Computing*, 30(6):1863–1920, 2000.

[13] X. Y. Li. *Optimization Algorithms for the Minimum-Cost Satisfiability Problem*. PhD thesis, Department of Computer Science, North Carolina State University, Raleigh, North Carolina, 2004. 162 pages.

[14] S. Liao and S. Devadas. Solving covering problems using LPR-based lower bounds. In *Proceedings of the 34th Design Automation Conference (DAC'97)*, pages 117–120, 1997.

[15] V. Manquinho, P. Flores, J. Marques-Silva, and A. Oliverira. Prime implicant computation using satisfiability algorithms. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI'99)*, pages 117–120, 1999.

[16] V. Manquinho and J. Marques-Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21:505–516, 2002.

[17] V. Manquinho and O. Roussel. Pseudo-Boolean evaluation 2005 `http://www.cril.univ-artois.fr/PB05/`, August 2005.

[18] J. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[19] S. Miyazaki, K. Iwama, and Y. Kambayashi. Database queries as combinatorial optimization problems. In *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications*, pages 448–454, 1996.

[20] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[21] S. Yang. Logic synthesis and optimization benchmarks user guide. Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, 1991.