Search Pruning Conditions for Boolean Optimization

Vasco M. Manquinho
vmm@algos.inesc.pt
Polytechnical Institute of Portalegre
Portalegre, Portugal

João Marques-Silva
jpms@inesc.pt
Technical University of Lisbon, INESC/CEL
Lisbon, Portugal

Abstract.

This paper proposes new algorithms for the Binate Covering Problem (BCP), a well-known restriction of Boolean Optimization. Binate Covering finds application in many areas of Computer Science and Engineering. In Artificial Intelligence, BCP can be used for computing minimum-size prime implicants of Boolean functions, of interest in Automated Reasoning and Non-Monotonic Reasoning. Moreover, Binate Covering is an essential modeling tool in Electronic Design Automation. The objectives of the paper are to briefly review branch-and-bound algorithms for BCP, to describe how to apply backtrack search pruning techniques from the Boolean Satisfiability (SAT) domain to BCP, and to illustrate how to strengthen those pruning techniques by exploiting the actual formulation of BCP. Experimental results, obtained on representative instances indicate that the proposed techniques provide significant performance gains for different classes of instances.

1 Introduction

The generic Boolean Optimization problem as well as several of its restrictions are well-known computationally hard problems, widely used as modeling tools in Computer Science and Engineering. These problems have been the subject of extensive research work in the past (see for example [1]). In this paper we address the Binate Covering Problem (BCP), one of the restrictions of Boolean Optimization. BCP can be formulated as the problem of finding a satisfying assignment for a given Conjunctive Normal Form (CNF) formula subject to minimizing a given cost function. As with generic Boolean Optimization, BCP also finds many applications, including the computation of minimum-size prime implicants, of interest in Automated Reasoning and Non-Monotonic Reasoning [12], and as a modeling tool in Electronic Design Automation (EDA) [4, 15].

In recent years, several powerful search pruning techniques have been proposed for solving BCP, allowing dramatic improvements in the ability to solving large and complex instances of BCP. (Details of the work on BCP can be found in [4, 8, 15].) Despite these improvements, and as with other NP-hard problems, additional search pruning ability allows in general very significant gains, both in the amount of search and in the run times. The ultimate consequence of proposing new pruning techniques is the potential ability for solving new classes of instances.

The main objective of this paper is to propose additional techniques for pruning the amount of search in branch-and-bound algorithms for solving covering problems. These techniques correspond to generalizations and extensions of similar techniques proposed in the Boolean Satisfiability (SAT) domain, where they have been shown to be highly effective [2, 14, 16]. In particular, and to our best knowledge, we provide for the first time conditions which enable branch-and-bound algorithms to backtrack *non-chronologically* whenever bounding due to the cost function is required to take place.

This paper is organized as follows. In Section 2 the notation used throughout the paper is introduced. Afterwards, branch-and-bound covering algorithms are briefly reviewed, giving emphasis to solutions based on SAT algorithms. In Section 4 we propose new techniques for reducing the amount of search. In particular we show how effective search pruning techniques from the SAT domain can be generalized and extended to the BCP domain. Experimental results are presented in Section 6, and the paper concludes in Section 7.

2 Preliminaries

An instance C of a covering problem is defined as follows,

$$\begin{array}{ll} \text{minimize} & \sum_{j=1}^{n} c_j \cdot x_j \\ \text{subject to} & A \cdot x \geq b, \quad x \in \{0,1\}^n \end{array} \tag{1}$$

where c_j is a non-negative integer cost associated with variable $x_j, 1 \leq j \leq n$ and $A \cdot x \geq b, x \in \{0,1\}^n$ denote the set of m linear constraints. If every entry in the $(m \times n)$ matrix A is in the set $\{0,1\}$ and $b_i = 1, 1 \leq i \leq m$, then C is an instance of the *unate covering problem* (UCP). Moreover, if the entries a_{ij} of A belong to $\{-1,0,1\}$ and $b_i = 1 - |\{a_{ij} : a_{ij} = -1, 1 \leq j \leq n\}|$, then C is an instance of the *binate covering problem* (BCP). Observe that if C is an instance of the binate covering problem, then each constraint can be interpreted as a propositional clause.

Conjunctive Normal Form (CNF) formulas are introduced next. The use of CNF formulas is justified by noting that the set of constraints of an instance C of BCP is equivalent to a CNF formula, and because some of the search pruning techniques described in the remainder of the paper are easier to convey in this alternative representation.

A propositional formula φ in *Conjunctive Normal Form* (CNF) denotes a boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$. The formula φ consists of a conjunction of propositional clauses, where each clause ω is a disjunction of literals, and a literal l is either a variable x_j or its complement $\bar{x}_j$. If a literal assumes value 1, then the clause is *satisfied*. If all literals of a clause assume value 0, the clause is *unsatisfied*. Clauses with only one unassigned literal are referred to as *unit*. Finally, clauses with more than one unassigned literal are said to be *unresolved*. In a search procedure, a *conflict* is said to be identified when at least one clause is unsatisfied. In addition, observe that a clause $\omega = (l_1 + \cdots + l_k), k \leq n$, can be interpreted as a linear inequality $l_1 + \cdots + l_k \geq 1$, and the complement of a variable x_j, $\bar{x}_j$, can be represented by $1 - x_j$.

When a clause is unit (with only one unassigned literal) an assignment can be implied. For example, consider a propositional formula φ which contains clause $\omega = (x_1 + \bar{x}_2)$ and assume that $x_2 = 1$. For φ to be satisfied, x_1 must be assigned value 1 due to ω. Therefore, we say that $x_2 = 1$ *implies* $x_1 = 1$ due to ω or that clause ω *explains* the assignment $x_1 = 1$. These logical implications correspond to the application of the unit clause rule [6] and the process of repeatedly applying this rule is called *boolean constraint propagation* [14, 16]. It should be noted that throughout the remainder of this paper some familiarity with backtrack search SAT algorithms is assumed. The interested reader is referred to the bibliography (see for example [1, 14] for additional references).

Covering problems are often solved by branch and bound algorithms [5, 8, 15]. In these cases, each node of the search tree corresponds to a selected unassigned variable and the two branches out of the node represent the assignment of 1 and 0 to that variable. These variables are named *decision variables*. The first node is called the *root* (or the top node) of the search tree and corresponds to the *first decision level*. The decision level of each decision is defined as one plus the decision level of the previous decision.

3 Search Algorithms for Covering Problems

The most widely known approach for solving covering problems is the classical branch and bound procedure [15], in which *upper bounds* on the value of the cost function are identified for each solution to the constraints, and *lower bounds* on the value of the cost function are estimated considering the current set of variable assignments. The search can be pruned whenever the lower bound estimation is higher than or equal to the most recently computed upper bound. In these cases we can guarantee that a better solution cannot be found with the current variable assignments and therefore the search can be pruned. The algorithms described in [5, 8, 15] follow this approach.

Several lower bound estimation procedures can be used, namely the ones based on linear-programming relaxations [8] or lagrangian relaxations [11]. Nevertheless, and for BCP, the approximation of a maximum independent set of clauses [4] is the most commonly used. The tightness of the lower bounding procedure is crucial for the algorithm's efficiency, because with higher estimates of the lower bound, the search can be pruned earlier. For a better understanding of lower bounding mechanisms, a method for approximating the maximum independent set of clauses is described in section 3.1. Covering algorithms also incorporate several powerful reduction techniques, a comprehensive overview of which can be found in [4, 15].

With respect to the application of SAT to Boolean Optimization, P. Barth [1] first proposed a SAT-based approach for solving pseudo-boolean optimization (i.e. a generalization of BCP). This approach consists of performing a linear search on the possible values of the cost function, starting from the highest, at each step requiring the next computed solution to have a lower cost than the most recently computed upper bound. Whenever a new solution is found which satisfies all the constraints, the value of the cost function is recorded as the current lowest computed upper bound. If the resulting instance of SAT is not satisfiable, then the solution to the instance of BCP is given by the last recorded solution.

Additional SAT-based BCP algorithms have been proposed. In [9] a different algorithmic organization is described, consisting in the integration of several features from SAT algorithms in a branch and bound procedure, *bsolo*, to solve the binate covering problem. The *bsolo* algorithm incorporates the most significant features from both approaches, namely the bounding procedure and the reduction techniques from branch and bound algorithms, and the search pruning techniques from SAT algorithms.

The algorithm presented in [9] already incorporates the main pruning techniques of the GRASP SAT algorithm [14]. Hence, *bsolo* is a branch and bound algorithm for solving BCP that implements a non-chronological backtracking search strategy, clause recording and identification of necessary assignments. Mainly due to an effective conflict analysis procedure which allows non-chronological backtracking steps to be identified, *bsolo* performs better than other branch and bound algorithms in several classes of instances, as shown in [9]. However, non-chronological backtracking is limited to one specific type of conflict. In section 4 we describe how to apply non-chronological backtracking to *all* types of conflicts. The main steps of a simplified version of the *bsolo* algorithm (see [10] for other details) can be described as follows:

1. Initialize the upper bound to the highest possible value as defined (i.e. given by $ub = \sum_{j=1}^{n} c_j + 1$).
2. Start by checking whether the current state yields a conflict. This is done by applying boolean constraint propagation and, in case a conflict is reached, by invoking the conflict analysis procedure, recording relevant clauses and proceeding with the search procedure or backtrack if necessary.
3. If a solution to the constraints has been identified, update the upper bound according to $ub = \sum_{j=1}^{n} c_j \cdot x_j$.
4. Estimate a lower bound given the current variable assignments. If this value is higher than or equal to the current upper bound, issue a bound conflict and bound the search by applying the conflict analysis procedure to determine which decision node to backtrack to. Continue from step 2.

3.1 Maximum Independent Set of Clauses

The estimation of lower bounds on the value of the cost function is a very effective method to prune the search tree and the accuracy of lower bounding procedures is critical for identifying areas of the search space where solutions to the constraints with lower values of the cost function cannot be found. This section reviews a commonly used greedy method to estimate a lower bound on the value of the cost function based on an independent set of clauses, which is also detailed for example in [4].

The greedy procedure consists of finding a set I of disjoint unate clauses, i.e. clauses with only positive literals and with no literals in common between them. Since maximizing the cost of I is a NP-hard problem, a greedy computation is used, as described in [5]. The effectiveness of this method largely depends on the clauses included in I. Usually, one chooses the clause which maximizes the ratio between its weight and its number of elements. The minimum cost for satisfying I is a *lower bound* on the solution of the problem instance and is given by, $Cost(I) = \sum_{\omega \in I} Weight(\omega)$ where $Weight(\omega) = \min_{x_j \in \omega} c_j$.

3.2 Bound Conflicts

In *bsolo* two types of conflicts which can be identified: *logical conflicts* that occur when at least one of the problem instance constraints becomes unsatisfied, and *bound conflicts* that occur when the lower bound is higher than or equal to the upper bound. When logical conflicts occur, the conflict analysis procedure from GRASP is applied and determines to which decision level the search should backtrack to (possibly in a non-chronological manner).

However, the other type of conflict is handled differently. In *bsolo*, whenever a bound conflict is identified, a new clause *must* be added to the problem instance in order for a logical conflict to be issued and, consequently, to bound the search. This requirement is inherited from the GRASP SAT algorithm where, for guaranteeing completeness, both conflicts and implied variable assignments *must* be explained in terms of the existing variable assignments [14]. With respect to conflicts, each recorded conflict clause is built using the assignments that are deemed responsible for the conflict to occur. If the assignment $x_j = 1$ (or $x_j = 0$) is considered responsible, the literal $\bar{x}_j$ (respectively, literal x_j) is added to the conflict clause. This literal basically states that in order to avoid the conflict one possibility is certainly to have instead the assignment $x_j = 0$ (respectively, $x_j = 1$). Clearly, by construction, after the clause is built its state is unsatisfied. Consequently, the conflict analysis procedure has to be called to determine to which decision level the algorithm must backtrack to. Hence the search is bound.

Whenever a bound conflict is identified, one possible approach to building a clause to bound the search would be to include all decision variables in the search tree. In this case, the conflict would always depend on the last decision variable. Therefore, backtracking due to bound conflicts would necessarily be chronological (i.e. to the previous decision level), hence guaranteeing that the algorithm would be complete. Suppose that the set $\{x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1\}$

corresponds to all the search tree decision assignments and ω_{bc} is the clause to be added due to a bound conflict. Then we would have $\omega_{bc} = (\bar{x}_1 + x_2 + x_3 + \bar{x}_4)$. Again, the problem with this approach (which was used in [9]) is that backtracking due to bound conflicts is always chronological, since it depends on all decisions made. In the following section we present a new procedure to build these clauses, which enable non-chronological backtracking due to bound conflicts.

4 SAT-Based Pruning Techniques for BCP

One of the main features of *bsolo* is the ability to backtrack non-chronologically when conflicts occur. This feature is enabled by the conflict analysis procedure inherited from the GRASP SAT algorithm. However, as illustrated in section 3.2, in the original *bsolo* algorithm non-chronological backtracking was only possible for logical conflicts. In the case of a bound conflict all the search tree decision assignments were used to explain the conflict. Therefore, these conflicts would always depend on the last decision level and backtracking would necessarily be chronological.

In this section we describe how to compute sets of assignments that explain bound conflicts. Moreover, we show that these assignments are not in general associated with all decision levels in the search tree; hence non-chronological backtracking can take place.

A bound conflict in an instance of the binate covering problem (BCP) *C* arises when the lower bound is equal to or higher than the upper bound . This condition can be written as $C.path + C.lower \geq C.upper$, where *C.path* is the cost of the assignments already made, *C.lower* is a lower bound estimate on the cost of satisfying the clauses not yet satisfied (as given for example by an independent set of clauses), and *C.upper* is the best solution found so far. From the previous equation, we can readily conclude that *C.path* and *C.lower* are the unique components involved in each bound conflict. (Notice that *C.upper* is just the lowest value of the cost function for the solutions of the constraints computed earlier in the search process.) Therefore, we will analyze both *C.path* and *C.lower* in order to establish the assignments responsible for a given bound conflict.

We start by studying *C.path*. Clearly, the variable assignments that cause the value of *C.path* to grow are solely those assignments with a value of 1. Hence, we can define a set of literals ω_{cp}, such that each variable in ω_{cp} has positive cost and is assigned value 1:

$$\omega_{cp} = \{l = \bar{x}_j : Cost(x_j) > 0 \wedge x_j = 1\} \tag{2}$$

which basically states that to decrease the value of the cost function (i.e. *C.path*) at least one variable that is assigned value 1 has instead to be assigned value 0.

We now consider *C.lower*. Let MIS be the independent set of clauses, obtained by the method described in section 3.1, that determines the value of *C.lower*. Observe that each clause in MIS is part of MIS because it is neither satisfied nor covered by some other clause in MIS. Clearly, for each clause $\omega_i \in MIS$ these conditions only hold due to the literals in ω_i that are assigned value 0. If any of these literals was assigned value 1, ω_i would certainly not be in MIS since it would be a satisfied clause. Consequently, we can define a set of literals that explain the value of *C.lower*:

$$\omega_{cl} = \{l : l = 0 \wedge l \in \omega_i \wedge \omega_i \in MIS\} \tag{3}$$

Now, as stated above, a bound conflict is solely due to the two components *C.path* and *C.lower*. Hence, this bound conflict will hold as long as the following clause ω_{bc} is unsatisfied:

$$\omega_{bc} = \omega_{cp} \cup \omega_{cl} \tag{4}$$

(Observe that the set union symbol in the previous equation denotes a disjunction of literals.) As long as this clause is unsatisfied, the values of *C.path* and *C.lower* will remain unchanged, and so the bound conflict will exist. We can thus use this unsatisfied clause ω_{bc} to analyze the bound conflict and decide where to backtrack to, using the conflict analysis procedure of GRASP [14]. We should observe that backtracking can be non-chronological, because clause ω_{bc} does not necessarily depend on all decision assignments. Moreover, due to the clause recording mechanism, ω_{bc} can be used later in the search process to prune the search tree. If these clauses would depend on all decision assignments, clause recording would not be used since the same set of decisions is never repeated in the search process.

Bound conflicts arise during the search process whenever we have $C.path + C.lower \geq C.upper$. Notice that when a new solution is found, $C.lower = 0$ because the independent set is empty (all clauses are satisfied) and *C.path* is equal to the cost of the new upper bound. Therefore, when we update *C.upper* with the new value, we have $C.path + C.lower = C.upper$ and a bound conflict is issued in order to backtrack in the search tree. These bound conflicts are just a particular case and the same process we described in this section is applied in order to build the conflict clause.

5 Reducing Dependencies in Bound Conflicts

As shown in the previous section, in BCP algorithms it is possible to establish conditions for implementing non-chronological backtracking due to bound conflicts. However, the ability to backtrack non-chronologically is strongly related with the ability for identifying a small set of assignments that explain each bound conflict. Sets of assignments that include many assignments irrelevant for actually explaining the bound conflict can drastically reduce the ability to backtrack non-chronologically. Hence, after computing explanations for bound conflicts, using the techniques described in the previous section, the next step is to identify assignments that can be discarded from each explanation by proving them irrelevant for the bound conflict to take place.

In this section we propose different techniques for reducing dependencies in the explanations of bound conflicts, hence reducing the number of literals in ω_{bc}.

5.1 Relating C.path and C.lower

Let l_j be a literal such that $l_j \in \omega_{cp}$ and $l_j \notin \omega_{cl}$. Then l_j is in ω_{bc} only due to the $C.path$ component explaining the bound conflict. Let MIS be the independent set, computed with the procedure described in [5], which is used to obtain the value of $C.lower$. In this situation, literal l_j can be removed from ω_{cp} provided the following conditions apply:

- There exists a satisfied clause ω_i such that $\bar{l}_j$ is the only literal which currently satisfies ω_i.
- All literals of ω_i besides l_j must be positive, unassigned and must not intersect MIS (so that ω_i can be added to MIS if l_j assumes value 0).
- All literals in ω_i must have a cost higher than or equal to the cost of literal l_j.
- No clause in MIS can contain l_j.

This reduction step can be made because if $l_j = 0$, ω_i would be in the independent set and the lower bound value would not decrease. Therefore, literal l_j can be deemed irrelevant for explaining the bound conflict and can be removed from ω_{bc}.

As an example, let us suppose that variables x_1, x_2 and x_3 belong to the cost function with the same cost and $x_1 = 1$. If a bound conflict occurs, from (2) $\bar{x}_1$ would be in ω_{bc}. However, suppose that clause $\omega_i = (x_1 + x_2 + x_3)$ is satisfied only due to x_1, i.e., x_2 and x_3 are unassigned. If x_2 and x_3 do not belong to any clause in MIS, $\bar{x}_1$ can be removed from ω_{bc} because $x_1 = 1$ is not relevant for the conflict. If variable x_1 was unassigned or assigned value 0, ω_i would be in MIS and the bound conflict would still occur.

It is interesting to observe that we can generalize the second condition, allowing ω_i to have positive literals whose variables are assigned value 0. Let us consider the example clause $\omega_i = (x_1 + x_2 + x_3 + x_4)$. Let $x_1 = 1$ and $x_2 = 0$. Moreover, let the cost of x_1

be no greater than the cost of x_2, let x_3, x_4 be such that ω_i would be in MIS if $x_1 = 0$, and let no other clause in MIS contain literal x_2. In this situation, the dependency on x_1 can be removed, and the dependency on x_2 need not be considered. Indeed, with $x_1 = 0$, ω_i would be in MIS and so the cost would not decrease. In addition, since the cost of x_2 is larger than or equal to the cost of x_1, by assigning value 1 to x_2, the cost would also not decrease. Hence the result follows. One should note that the same reasoning applies for an *arbitrary* number of variables assigned value 0 in a given clause with a single literal assigned value 1.

Next we show how ω_{cl} can be simplified by evaluating the consequences of modifying the value of some literals on the value of *C.path*. Suppose we have a literal $l = x_j$, with $l \in \omega_{cl}$ and let $x_j = 0$. If x_j only belongs to one clause ω_i of the independent set and its cost is greater than or equal to the minimum cost of ω_i, then l can be removed from ω_{bc}. To better understand how this is possible, suppose instead that $x_j = 1$. In this situation, ω_i would not be in the independent set (it would be a satisfied clause) and the $C.lower$ component would be lower[1]. However, since the cost of the variable is higher than or equal to the minimum cost of ω_i, the $C.path$ component would be higher, and hence the conflict would still hold. So, the assignment $x_j = 0$ is irrelevant for the conflict to arise and literal l can be removed from ω_{bc}. Observe that even if a clause ω_i, containing a literal $x_j = 0$, also contains other literals assigned value 0 (e.g. $x_k = 0$), the same reasoning still applies, and dependency on x_j can be removed. This holds even when $x_k = 0$ is contained in more than one clause of MIS.

Another reduction technique consists of using a satisfied clause to reduce a dependency from ω_{cl}. Let us consider the following set of clauses,

$$\begin{array}{rcl}\omega_1 & = & (x_1 + x_2 + x_3) \\ \omega_2 & = & (x_1 + x_4 + x_5) \\ \omega_3 & = & (\overline{x_1} + x_3 + x_4)\end{array} \tag{5}$$

with $x_1 = 0$, x_2, x_3, x_4, x_5 unassigned and ω_1 and ω_2 be part of MIS. Let the cost of x_2, x_3, x_4, x_5 be less than or equal to the cost of x_1. Finally, let no other clause in MIS contain x_1. If x_1 would take value 1, $C.lower$ would decrease by 1 since ω_1 and ω_2 would be satisfied, but ω_3 would now be in MIS. However, $C.path$ would be raised due to the cost of x_1 and the conflict would still hold. Hence, the dependency on x_1 can be removed.

5.2 Using Excess Cost Value

Let us consider a bound conflict and let $diff = (C.path + C.lower) - C.upper$. Clearly, $diff \geq 0$.

It is plain that if $C.path$ was lower by $diff$, the bound conflict would still hold since we would then have $C.upper = C.path + C.lower$. Therefore, we may conclude that not all assignments in $C.path$ are necessary for explaining the conflict, since if some assignments were not made, we would still have a bound conflict. In this case, it is possible to remove some literals from ω_{cp} as long as their cost is lower than or equal to $diff$.

Moreover, the value of $diff$ can also be used for reducing dependencies from $C.lower$. Notice that if we remove a subset of clauses D_MIS from MIS (used to obtain $C.lower$) such that,

$$Cost(D\_MIS) \leq diff \qquad \text{where} \tag{6}$$

$$Cost(D\_MIS) = \sum_{\omega \in D\_MIS} Weight(\omega) \tag{7}$$

then the lower bound conflict will still hold since $C.upper \leq C.path + C.lower$, where $C.lower$ is now obtained from the independent set of clauses $MIS \setminus D\_MIS$. Therefore, the lower bound conflict clause ω_{bc} can still be built using (4), but the ω_{cl} can now be reformulated as

$$\omega_{cl} = \{l : l = 0 \wedge l \in \omega_i \wedge \omega_i \in MIS \setminus D\_MIS\} \tag{8}$$

Moreover, the simplifications described above for ω_{cl} can now be applied to the resulting ω_{cl}.

One should note that the reduction on the number of dependencies relies on which clauses we choose to include in D_MIS. If a clause from MIS is selected with assigned literals belonging to ω_{bc} because of other clauses in MIS or due to ω_{cp}, then the dependencies are exactly the same. Therefore, it is desirable that D_MIS be a subset of MIS such that the number of dependencies in ω_{bc} be minimum. A greedy procedure is used for selecting the clauses to remove from MIS.

5.3 Resolution-Induced Dependency Reduction

In this section we illustrate how the resolution operation [13] can be used for establishing conditions that permit the elimination of dependencies. We should note that the proposed conditions, even though based on the resolution operation, do not require the explicit creation of new clauses.

The conditions proposed subsequently can be applied for removing dependencies from ω_{cp} and ω_{cl}. In all cases, we use examples to illustrate the application of resolution, but provide the necessary conditions for generic application.

We start by studying simplifications to ω_{cp} established with the resolution operation. Let us consider the following set of clauses,

$$\begin{array}{rcl}\omega_1 & = & (x_1 + x_2 + x_3) \\ \omega_2 & = & (\overline{x_1} + x_2 + x_4)\end{array} \tag{9}$$

with $x_2 = 1$, and such that x_3, x_4 are not covered by the currently computed MIS. x_1 can either be assigned or unassigned, and can either be or not be covered by the currently computed MIS. By applying resolution between ω_1 and ω_2, with respect to x_1, we obtain the resulting clause $\omega_3 = c(\omega_1, \omega_2, x_1) = (x_2 + x_3 + x_4)$. Now, ω_3 is certainly satisfied solely by x_2. Hence, we can conclude that the dependency on x_2 can be removed by applying the previous results on simplifying ω_{cp}. Notice that x_1 can be *any* variable. However, if x_1 is unassigned and not covered by MIS, then we can immediately apply the previous results on simplifying ω_{cp}.

Next, we illustrate one additional form of using the resolution operation for removing dependencies. As an example, assume a bound conflict, and consider the following set of clauses,

$$\begin{array}{rcl}\omega_1 & = & (x_1 + x_2 + x_3) \\ \omega_2 & = & (\overline{x_1} + x_4 + x_5)\end{array} \tag{10}$$

where x_1 is assigned either value 0 or 1, its cost is 0, and such that the dependency on x_1 is only due to ω_1 or ω_2. Furthermore, let us assume that ω_1 would be part of MIS with $x_1 = 0$, and that ω_2 would be part of MIS with $x_1 = 1$. In this situation the dependency on x_1 can be removed. Notice that if the cost of x_1 is non-zero, then the removal of the dependency on x_1 is guaranteed by the previous results (section 5.1) on simplifying ω_{cl}.

Clearly, the application of the resolution operation can be generalized and used for eliminating more than one variable, the only drawback being the computational effort involved.

6 Experimental Results

In this section we compare different algorithms for solving BCP on example instances taken from digital circuit testing problems [7]. Due to space limitations, only the most representative instances are presented. For the experimental results given below, the CPU times were obtained on a SUN Sparc Ultra I, running at 170MHz, and with 100 MByte of physical memory. In all cases the maximum CPU time that each algorithm was allowed to spend on any given

[1] In fact, if the $C.lower$ would be recomputed all over again, it is not guaranteed that it would decrease. Nevertheless, we know that without clause ω_i satisfied by $x_j = 1$, $MIS \backslash \{\omega_i\}$ it is still an independent set of clauses. Therefore, $MIS \backslash \{\omega_i\}$ can be used as a *low* estimate of $C.lower$.

instance was 1 hour. The experimental procedure consisted of running a selected set of problem instances with the *bsolo* algorithm, as described in Sections 3 and 4. These results are shown in Table 1. Next, in Table 2, we present the results of *bsolo* with the dependency elimination techniques of Section 5. For both tables, `Dec`, `NCB`, and `JMP` denote, respectively, the total number of decisions, number of non-chronological backtracks and largest backjump in the search tree, whereas `time` and `mem` indicate, respectively, that the time and memory limits were reached.

The experimental results from Tables 1 and 2 clearly indicate the efectiveness of the proposed techniques with redutions in both the time spent to solve the problem instances and the number of decisions. A more efective pruning can also be observed, with an increase in the number of non-chronological backtracks and larger jumps in the search tree.

Finally, in Table 3 we can observe the results of several other algorithms on the same set of instances. Clearly, *lp_solve* [3] (a generic Integer Linear Programming solver) is unable to solve almost all instances given the time limit. *scherzo* [5], a state of the art BCP solver, which incorporates several powerful pruning techniques in a classical branch-and-bound algorithm, is also unable to solve most of the example instances. The SAT-based linear search algorithm *opbdp* [1] is able to solve most instances, hence suggesting that these instances are well-suited for SAT-based solvers. Notice however that *bsolo* is faster than *opbdp* in most examples, and in some cases the improvement exceeds 1 order magnitude.

		bsolo			
Benchmark	min.	CPU	Dec.	NCB	Jump
c1908_F469@0	–	ub13	117079	721	9
c3540_F20@1	6	1045.14	3359	218	7
c432_F1gat@1	8	575.16	14756	608	53
c432_F37gat@1	9	ub15	218136	35785	21
c499_Fic2@1	–	ub41	1003200	1586	3
c6288_F35gat@1	4	107.69	756	41	42
c6288_F69gat@1	6	1413.17	4048	110	41
9symml_F6@0	9	6.05	272	23	4
alu4_Fj@0	6	185.59	1292	55	4
alu4_Fl@1	6	146.01	999	81	4
apex2_Fv14@1	10	20.15	908	48	4
apex2_Fv17@1	12	23.38	1082	70	5
duke2_Fv7@0	5	13.31	335	33	12
misex3_Fa@0	9	56.78	898	83	14
misex3_Fb@1	8	83.91	1038	71	8
spla_Fv14@0	8	38.93	914	120	12

Table 1. Results for bsolo

		bsolo			
Benchmark	min.	CPU	Dec.	NCB	Jump
c1908_F469@0	–	ub13	111386	1057	7
c3540_F20@1	6	907.40	2939	213	7
c432_F1gat@1	8	541.48	14117	647	53
c432_F37gat@1	9	ub14	286490	48534	21
c499_Fic2@1	–	ub41	1003200	1586	3
c6288_F35gat@1	4	44.42	555	39	42
c6288_F69gat@1	6	608.99	2198	94	41
9symml_F6@0	9	6.12	272	23	4
alu4_Fj@0	6	145.73	1034	46	5
alu4_Fl@1	6	132.75	933	73	5
apex2_Fv14@1	10	20.41	936	60	4
apex2_Fv17@1	12	23.60	1058	78	5
duke2_Fv7@0	5	12.93	332	32	12
misex3_Fa@0	9	55.18	879	81	14
misex3_Fb@1	8	80.47	1006	69	8
spla_Fv14@0	8	28.23	785	113	10

Table 2. Results for bsolo with dependency reductions

7 Conclusions

This paper extends well-known search pruning techniques, from the Boolean Satisfiability domain, to branch-and-bound algorithms for solving the Binate Covering Problem. The paper also describes conditions that allow for non-chronological backtracking in the presence of bound conflicts. To our best knowledge, this is the first time that branch-and-bound algorithms are augmented with the ability for backtracking non-chronologically in the presence of conflicts that result from bound conditions. In addition, we have established conditions for reducing the size of bound conflict explanations, which further elicits non-chronological backtracking.

		Algorithms			
Benchmark	min.	*lp_solve*	*scherzo*	*opbdp*	*bsolo*
c1908_F469@0	–	time	time	ub 24	ub13
c3540_F20@1	6	time	mem.	ub 13	907.40
c432_F1gat@1	8	ub 15	time	1148.27	541.48
c432_F37gat@1	9	time	time	3574.44	ub14
c499_Fic2@1	–	time	time	ub 41	ub41
c6288_F35gat@1	4	time	mem.	1330.95	44.42
c6288_F69gat@1	6	time	mem.	ub 9	608.99
9symml_F6@0	9	ub 9	29.44	1.59	6.12
alu4_Fj@0	6	time	879.05	413.71	145.73
alu4_Fl@1	6	time	1638.98	557.14	132.75
apex2_Fv14@1	10	ub 10	mem.	624.07	20.41
apex2_Fv17@1	12	time	mem.	532.94	23.60
duke2_Fv7@0	5	time	mem.	18.20	12.93
misex3_Fa@0	9	time	mem.	182.41	55.18
misex3_Fb@1	8	time	mem.	983.55	80.47
spla_Fv14@0	8	time	mem.	215.79	28.23

Table 3. Algorithm comparison

Preliminary results obtained on several instances of the Binate Covering Problem indicate that the proposed techniques are indeed effective and can be significant for specific classes of instances, in particular for instances of covering problems with sets of constraints that are hard to satisfy.

REFERENCES

[1] P. Barth. A Davis-Putnam Enumeration Algorithm for Linear Pseudo-Boolean Optimization. Technical Report MPI-I-95-2-003, Max Plank Institute for Computer Science, 1995.

[2] R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, 1997.

[3] M. R. C. M. Berkelaar. UNIX Manual Page of lp-solve. Eindhoven University of Technology, Design Automation Section, ftp://ftp.es.ele.tue.nl/pub/lp_solve, 1992.

[4] O. Coudert. Two-Level Logic Minimization, An Overview. *Integration, The VLSI Journal*, vol. 17(2):677–691, October 1993.

[5] O. Coudert. On Solving Covering Problems. In *Proceedings of the ACM/IEEE Design Automation Conference*, June 1996.

[6] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, vol. 7:201–215, 1960.

[7] P. F. Flores, H. C. Neto, and J. P. M. Silva. An exact solution to the minimum-size test pattern problem. In *Proceedings of the IEEE International Conference on Computer Design*, pages 510–515, October 1998.

[8] S. Liao and S. Devadas. Solving Covering Problems Using LPR-Based Lower Bounds. In *Proceedings of the ACM/IEEE Design Automation Conference*, 1997.

[9] V. M. Manquinho, P. F. Flores, J. P. M. Silva, and A. L. Oliveira. Prime implicant computation using satisfiability algorithms. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, pages 232–239, November 1997.

[10] V. M. Manquinho and J. P. Marques-Silva. On solving boolean optimization with satisfiability-based algorithms. In *Sixth International Symposium on Artificial Intelligence and Mathematics*, January 2000.

[11] G. L. Nemhauser and L. A. Wosley. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[12] C. Pizzuti. Computing Prime Implicants by Integer Programming. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, November 1996.

[13] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1994.

[14] J. P. M. Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[15] T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Explicit and Implicit Algorithms for Binate Covering Problems. *IEEE Transactions on Computer Aided Design*, vol. 16(7):677–691, July 1997.

[16] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.