On Solving Boolean Optimization with Satisfiability-Based Algorithms

Vasco M. Manquinho João Marques-Silva

vmm@algos.inesc.pt jpms@inesc.pt

Department of Informatics
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. Binate Covering is also an essential modeling tool in Electronic Design Automation (EDA). The objectives of the paper are to briefly review algorithmic solutions for BCP, and to describe how to apply search pruning techniques from the Boolean Satisfiability (SAT) domain to BCP. Furthermore, we generalize these pruning techniques, in particular the ability to backtrack non-chronologically, to exploit the actual formulation of the binate covering problem. 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 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 [16], and as a modeling tool in Electronic Design Automation (EDA) [11, 14].

In recent years, several powerful algorithmic techniques have been proposed for solving BCP, allowing dramatic improvements in the ability to solving large and complex instances of BCP[1]. Despite these improvements, and as with other NP-hard problems, new effective techniques allow in general very significant gains, both in the amount of search and in the run times. The ultimate consequence of proposing new algorithmic 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, 17, 19]. 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 upper and lower bound conditions require bounding 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 5, and the paper concludes in Section 6.

[1] Several examples of these techniques can be found in [4, 5, 10, 12].

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 linear constraints. If every entry $(m \times n)$ of 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* (BCP). Moreover, if the entries a_{ij} of A belong to $\{-1,0,1\}$ and $b_i = 1 - |\{a_{ij} : a_{ij} = -1\}|$, then C is an instance of the *binate covering problem* (BCP). It is interesting to 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. Because 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. We should also 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 instance, 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 [7] and the process of repeatedly applying this rule is called *boolean constraint propagation* [17]. We should note 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, 17] for additional references).

Covering problems are often solved by branch and bound algorithms [4, 10, 18]. 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*. Hence, the top nodes define the first decision levels of the search tree.

3 Backtrack Search Algorithms for Covering Problems

The most widely known approach for solving covering problems is the classical branch and bound procedure [11], 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 [4, 12, 18] follow this approach.

There are several lower bound estimation procedures that can be used, namely the ones based on linear-programming relaxations [12] or lagrangian relaxations [15], but the approximation of a maximum independent set of clauses [5] is the most commonly used one. 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 of approximation of a maximum independent set of clauses is described in section 3.3. Moreover, in recent years several powerful problem instances simplification techniques have been proposed. See [3, 4, 5, 9, 18] for detailed description of these techniques.

In the next few sections we briefly review alternative approaches for solving BCP, which are known to be competitive for specific types of instances, e.g. when the constraints are hard to solve. These approaches, namely the ones based on boolean satisfiability algorithms, have different pruning strategies which are not commonly used in branch and bound algorithms for solving BCP. In section 3.2 an algorithm which combines features from both approaches is described.

```
int min_prime($\varphi$) {
  $ub = \sum c_j$;
  while ($ub \geq 0$) {
    $\varphi = \varphi \cup \{\sum c_j \cdot x_j < ub\}$;
    status = solve_sat($\varphi$);
    $\varphi = \varphi - \{\sum c_j \cdot x_j < ub\}$;
    if (status == SATISFIABLE)
      $ub = \sum c_j \cdot x_j$;
    else break;
  }
  return $ub$;
}
```

Figure 1: SAT-based linear search algorithm

3.1 SAT-Based Linear Search Algorithm

In [1] P. Barth describes how to solve pseudo-boolean optimization (i.e. a generalization of BCP) using a propositional satisfiability (SAT) algorithm. However, the algorithm described in [1] is based on the Davis-Putnam [7] procedure, which has been shown to be particularly inefficient for a large number of instances of SAT. In [13], a new algorithm based on the GRASP SAT algorithm [17] is proposed, which is able to obtain better experimental results. Both these two algorithms interpret the binate covering problem (BCP) as a SAT problem defined by the constraints $A \cdot x \geq b$, but with the additional constraint of having to find a solution with lower cost than an upper bound value. The possible values assumed by the cost function for the different assignments to the problem variables $\{x_1, \ldots, x_n\}$ range from 0, when all variables are assigned value 0, to $\sum_{j=1}^{n} c_j$, when all variables with $c_j > 0$ are assigned value 1. Initially, the upper bound *ub* on the value of the cost function is given by:

$$ub = \sum_{j=1}^{n} c_j + 1 \tag{2}$$

SAT-based linear search algorithms perform a linear search on the possible values of the cost function, starting from the highest (given by (2)), at each step requiring the next computed solution to have a cost less than the most recently computed upper bound. Whenever a new solution is found which satisfies all the constraints, the upper bound *ub* is updated to:

$$ub = \sum_{j=1}^{n} c_j \cdot x_j \tag{3}$$

If the resulting instance of SAT is not satisfiable, then the solution to the instance of BCP is given by ub. Starting with the ub given by (2), SAT-based linear search algorithms consist of applying the following steps (see fig 1):

1. Create a new constraint $\sum_{j=1}^{n} c_j \cdot x_j < ub$. This inequality basically requires that a computed solution must have a lower cost than the best one found so far.

2. Solve the resulting instance of a satisfiability problem, defined on linear inequalities. The modification of most SAT algorithms to deal with this generalization is straightforward.

3. If the instance is satisfiable, then update ub according to (3) and go back to 1. Otherwise, the solution to the covering problem is ub. In those cases where the initial upper bound is never updated, the problem does not have a solution.

3.2 SAT-Based Branch and Bound Algorithm

Additional SAT-based BCP algorithms have been proposed. In [13] a new 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. This new framework from *bsolo* incorporates the main features from both approaches, namely the bounding procedure and reduction techniques from branch and bound algorithms, and search pruning techniques from SAT algorithms.

Originally, the algorithm presented in [13] already incorporated the main pruning techniques of the GRASP SAT algorithm [17]. To our knowledge, *bsolo* was the first branch and bound algorithm for solving BCP that implemented

```
int bsolo($\varphi$) {
  $ub = \sum c_j + 1$;
  while (TRUE) {
    if (!reduce_problem())
      return $ub$;
    identify_partitions();
    decide();
    if (!consistent_state())
      return $ub$;
    while (Estimate_LB() $\geq$ $ub$) {
      Issue_LB_based_conflict();
      if (!consistent_state())
        return $ub$;
    }
  }
}
int consistent_state() {
  do {
    while (Deduce() == CONFLICT)
      if (diagnose() == CONFLICT)
        return FALSE;
    apply_deduction = FALSE;
    if (Solution_found()) {
      Update_ub();
      Issue_UB_based_conflict();
      apply_deduction = TRUE;
    }
  } while (apply_deduction);
  return TRUE;
}
```

Figure 2: SAT-based branch and bound algorithm

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 [13]. However, non-chronological backtracking was 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 the algorithm (see fig. 2) can be described as follows:

1. Initialize the upper bound to the highest possible value as defined in (2).
2. Apply function *reduce_problem* to reduce the problem instance dimension by applying the techniques from standard branch and bound covering algorithms. Afterwards, identify problem partitions and branch on a given decision variable (i.e. make a decision assignment).
3. The function *consistent_state* checks 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.
4. If a solution to the constraints has been identified, update the upper bound according to (3) and issue an upper bound conflict to backtrack on the search tree. (Observe that the only way to reduce the value of the current solution is to backtrack with the objective of finding a solution with a lower cost.)
5. Estimate a lower bound given the current variable assignments. If this value is higher than or equal to the current upper bound, issue a lower bound conflict and bound the search by applying the conflict analysis procedure to determine the node to backtrack to (using function *consistent_state*). Continue the search from step 2.

3.3 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

```
maximal_independent_set($\varphi$) {
  MIS = empty set;
  do{
    $\omega$ = choose_clause($\varphi$);
    MIS = MIS $\cup$ $\{\omega\}$;
    $\varphi$ = delete_intersecting_clauses($\varphi$,$\omega$);
  } while ($\varphi$ not empty);
  return MIS;
}
```

Figure 3: Algorithm for computing a MIS

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 independent set of clauses, which is also detailed for example in [3].

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 shown in fig. 3. 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 value of the problem instance and is given by,

$$Cost(I) = \sum_{\omega \in I} Weight(\omega) \quad \text{where} \tag{4}$$

$$Weight(\omega) = \min_{x_j \in \omega} c_j \tag{5}$$

3.4 Upper and Lower Bound Conflicts

In *bsolo* there are three types of conflicts which can arise: *logical conflicts* that occur from the problem constraints, *upper bound conflicts* that occur when a solution to the constraints is found, and *lower bound conflicts* that take place 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 two types of conflicts are treated differently. In *bsolo*, whenever we have an upper or lower bound conflict, 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 [17]. With respect to conflicts, each recorded conflict clause is built using the assignments that are deemed responsible for the conflict to arise. 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 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.

We start by studying upper bound conflicts. In these situations, one possible approach to build 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 upper bound conflicts would necessarily be chronological (i.e. to the previous decision level), hence guaranteeing that the algorithm would be complete.

The previous strategy can also be used for lower bound conflicts. By building a clause involving all decision assignments present in the search tree, we guarantee that the search is bound and ensure that the algorithm is 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_{lb} is the clause to be added due to a lower bound conflict. Then we would have $\omega_{lb} = (\bar{x}_1 + x_2 + x_3 + \bar{x}_4)$. Again, the problem with this approach (which was used in [13]) is that backtracking is always chronological, since it depends on all decisions made. In sections 4.1 and 4.2 we will present new ways for building these clauses, which enable non-chronological backtracking due to upper and lower 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 3.4, in the original *bsolo* algorithm non-chronological backtracking was only possible for logical conflicts. In the case of an upper or lower bound conflict all the search tree decision assignments were used to explain the conflict. Therefore, these conflicts would depend on the last decision level and backtracking would always be chronological.

In this section we describe how to compute sets of assignments that explain upper and lower 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.

4.1 Dependencies in Upper Bound Conflicts

As mentioned in Section 3.4, upper bound conflicts correspond to the process of bounding the search when a new solution (with lower cost) is found. In *bsolo*, and because of the conflict analysis procedure, the bounding process requires creating a new conflict clause. Moreover, all decision variables were present in this clause, thus preventing non-chronological backtracks from occurring. However, it is straightforward to conclude that the assignments which characterize the computed solution are the ones that allow the value of the cost function to grow, i.e., the assignments of 1 to variables with positive cost in the cost function. Therefore, we should backtrack to a decision level where at least one of these assignments is toggled to its complemented value. Let ω_{ub} be the clause added due to an upper bound conflict. This clause is defined by:

$$\omega_{ub} = \{l = \bar{x}_j : Cost(x_j) > 0 \wedge x_j = 1\} \tag{6}$$

Consequently, it becomes possible to backtrack non-chronologically after identifying an upper bound conflict. This is illustrated next.

Let $f(x_1, x_2, x_3, x_4) = x_1 + x_2 + x_3$ be the cost function to minimize, and the set of constraints be:

$$(x_1 + x_4) \cdot (\bar{x}_4 + \bar{x}_2) \cdot (x_3 + x_4) \cdot (\bar{x}_3 + x_4) \tag{7}$$

Let us assume the sequence of decision assignments $x_1 = 1$ and $x_2 = 0$. Suppose that the next decision assignment is $x_3 = 1$, that implies $x_4 = 1$. Then all clauses are satisfied, and the value of the cost function is 2. Next, an upper bound conflict is issued, the clause $\omega_{ub} = (\bar{x}_1 + \bar{x}_3)$ is created (observe that the assignment $x_2 = 0$ is irrelevant for being able to reduce the current upper bound estimate) and decision $x_3 = 1$ is erased. Afterwards, the assignment $x_3 = 0$ is implied (from ω_{ub}), which again implies $x_4 = 1$ from (7), thus satisfying all clauses. In this case the value of the cost function is 1. Now, since the value of x_3 is implied, we can readily create a new upper bound conflict clause $(\bar{x}_1)$, which indicates that we should backtrack *immediately* to the decision stage where the assignment $x_1 = 1$ is defined. Hence, we backtrack non-chronologically, skipping the backtrack to the decision assignment $x_2 = 0$.

4.2 Dependencies in Lower Bound Conflicts

A lower bound conflict in a binate covering problem (BCP) *C* arises when the lower bound is equal to or higher than the upper bound and we can write this condition as follows:

$$C.path + C.lower \geq C.upper \tag{8}$$

where *C.path* is the cost of the assignments already made, *C.lower* is a lower bound estimate on the cost for satisfying the clauses still not satisfied, 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 lower bound conflict. (Notice that *C.upper* is just the value of the cost function for a solution computed earlier in the search process.) Therefore, we will analyze both *C.path* and *C.lower* components in order to establish the assignments responsible for a given lower 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:

$$\omega_{cp} = \{l = \bar{x}_j : Cost(x_j) > 0 \wedge x_j = 1\} \tag{9}$$

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. (Observe that ω_{cp} is equivalent to ω_{ub}, as described in Section 4.1.)

We now consider *C.lower*. Let *MIS* be the independent set of clauses, obtained by the method described in section 3.3, that determines the value of *C.lower*. Note 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 ω_i 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*:

$$\omega_{cl} = \{l : l = 0 \wedge l \in \omega_i \wedge \omega_i \in MIS\} \tag{10}$$

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

$$\omega_{lb} = \omega_{cp} \cup \omega_{cl} \tag{11}$$

(Observe that the set union symbol in the previous equation denotes conjunction of clauses.) As long as this clause is unsatisfied, the values of *C.path* and *C.lower* will remain unchanged, and so the lower bound conflict will exist. We can thus use this unsatisfied clause ω_{lb} to analyze the lower bound conflict and decide where to backtrack to, using the conflict analysis procedure of GRASP [17]. We should observe that backtracking can be non-chronological, because clause ω_{lb} does not necessarily depend on all decision assignments.

With respect to (11) a more careful analysis allows us to conclude that not all of these literals are necessary. Suppose that the lower bound is higher than the upper bound and define this difference as $diff = (C.path + C.lower) - C.upper$. It is clearly true that if $C.path$ was lower by $diff$, the lower bound conflict would still hold because we would have $C.upper = C.path + C.lower$. Therefore, we may conclude that not all assignments in $C.path$ are necessary to explain the conflict, since if some assignments were not made, we would still have a lower bound conflict. In this case, it is possible to remove some literals from ω_{cp} such that their cost is lower than or equal to $diff$.

In order to implement this technique, one interesting problem is to decide which literals should be removed from ω_{cp}. In *bsolo* an heuristic procedure is used for removing the literals that have been assigned at the most recent levels of the decision tree. Consequently, the likelihood of backtracking non-chronologically is higher, since these conflicts will be more dependent on the earlier levels of the search tree. Notice that if a literal l is removed from ω_{cp}, but if $l \in \omega_{cl}$ to explain the value in $C.lower$, then we must have $l \in \omega_{lb}$ and there is no reduction in the dependencies of the conflict clause.

Moreover, it is interesting to observe that a clause resulting from a lower bound conflict can be simpler. We have only described how simplifications can be made to the *C.path* component, but other simplifications can also be applied to the literals added due to the independent set of clauses (MIS). 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 higher than or equal to the minimum cost of ω_i, then l can be removed from ω_{lb}. 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 [2]. 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_{lb}.

4.3 Handling Reduction Techniques

As mentioned in the previous sections, for implementing non-chronological backtracking each implied variable assignment needs to be properly explained in order to guarantee that the resulting branch-and-bound algorithm is complete. Consequently, it is necessary that, whenever there is a variable assignment implied due to the application of a reduction technique (e.g., variable dominance, limit lower bound theorem, etc.), a new clause is built and added to the problem instance as an explanation for that assignment. Clearly, we could create this new clause by using all decision assignments in the decision tree, but this would negatively affect the ability of the search algorithm to backtrack non-chronologically. As before, we must identify conditions for using a reduced set of assignments instead of all decision assignments. In this section we illustrate how this is done for assignments implied due to the application of the limit lower bound theorem. For the other reduction techniques, a similar approach is used.

The limit lower bound theorem [4] is applied to a variable x_j whenever,

$$C.upper - (C.path + C.lower) \leq Cost(x_j) \tag{12}$$

[2] 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 satisfied by $x_j = 1$, $MIS \backslash \{\omega\}$ it is still an independent set of clauses. Therefore, $MIS \backslash \{\omega\}$ can be used as a low estimation of $C.lower$.

		lp-solve	**cplex**	**scherzo**	**opbdp**	**min-prime**	**bsolo**
Benchmark	min.	CPU	CPU	CPU	CPU	CPU	CPU
aim-100-1_6-yes1-2	100	–	–	–	1104.5	0.01	0.28
aim-100-3_4-yes1-4	100	–	–	11.56	0.19	0.15	0.68
aim-100-6_0-yes1-1	100	–	295.2	0.87	0.02	0.05	0.28
aim-200-1_6-yes1-3	200	–	–	–	–	0.05	0.41
aim-200-3_4-yes1-1	200	–	–	–	9.60	0.22	2.86
aim-200-6_0-yes1-2	200	–	–	184.18	1.13	1.79	1.71
aim-50-1_6-yes1-1	50	757.3	113.4	0.76	0.02	0.01	0.06
aim-50-2_0-yes1-2	50	1284.5	107.6	1.81	0.09	0.01	0.11
aim-50-3_4-yes1-3	50	86.4	62.2	0.18	0.01	0.02	0.09
ii8a1	54	162.8	63.0	0.33	0.62	63.0	0.52
ii8a2	–	ub 149	ub 147	–	ub 141	ub 150	ub 140
ii8b1	191	ub 243	840.4	–	ub 191	ub 191	1042.19
ii8c1	–	ub 364	ub 304	–	ub 302	ub 302	ub 302
ii8d1	–	–	ub 367	–	–	ub 343	ub 343
jnh1	92	–	ub 93	20.47	0.59	6.90	4.49
jnh12	94	–	2251.7	0.87	0.01	0.16	0.25
jnh7	89	ub 89	ub 90	1.49	0.10	3.60	0.63
ssa7552-038	1148	–	ub 1449	–	ub 1452	ub 1449	109.20
ssa7552-159	1327	ub 1327	ub 1327	14.16	ub 1327	ub 1327	4.58
ssa7552-160	1359	ub 1359	ub 1359	–	ub 1359	ub 1359	9.26

Table 1: Algorithm comparison

In these cases, the assignment $x_j = 0$ is implied.

Let ω_{llb} be a clause that must be added in order to explain the assignment $x_j = 0$, which is implied by applying the limit lower bound theorem. Notice that this theorem is applied because of the values of $C.path$ and $C.lower$. Thus, the assignments that explain these two values are also the explanation sought for the assignment $x_j = 0$. Therefore, clause ω_{llb} is constructed as follows,

$$\omega_{llb} = \omega_{cp} \cup \omega_{cl} \cup \{\bar{x}_j\} \tag{13}$$

where ω_{cp} and ω_{cl} are the literals which explain the values in $C.path$ and $C.lower$, as described in section 4.2. Therefore, ω_{llb} becomes a new unit clause and consequently implies the assignment $x_j = 0$. (Hence, we say that the assignment $x_j = 0$ is explained by ω_{llb}.)

5 Experimental Results

In this section we compare different algorithms for solving BCP on two different sets of instances. The first set, whose results are shown in table 1, consists of instances from the minimum-size prime implicant problem for Boolean functions. These instances were obtained from satisfiable instances of the DIMACS benchmark set [6], using the model described in [13, 16]. The second set, whose results are shown in all subsequent tables, consists of instances from the minimum-size test pattern problem [8].

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 instance was 1 hour.

Whenever an algorithm was not able to find the optimum value for a given problem instance, the best computed upper bound is shown (provided the algorithm was able to compute one). In some situations, the reason for the algorithm to abort is shown. This can be because the time limit was reached or because the available memory was not enough.

In table 1 we present experimental results for several algorithms for instances of the minimum-size prime implicant problem. This comparison involves the general purpose linear programming tools *lp-solve* and commercial optimization tool *cplex*, the BCP branch-and-bound algorithm *scherzo* [5], the SAT-based linear search algorithms *opbdp* [1] and *min-prime* [13], and the proposed algorithm *bsolo*. We should note that *scherzo* was specifically developed for solving BCP and includes several powerful search pruning techniques.

The results in this table clearly show how ineffective general purpose algorithms are for solving the minimum-size prime implicant problem, and support the conclusion that more dedicated algorithms may well be the most adequate choice. It should be observed that, in spite of the problem reduction techniques that it incorporates, *scherzo* performs very poorly. The main reason for this fact is that the techniques *scherzo* incorporates are more suited for solving less

bsolo		**no LB explanation**				**LB explanation**			
Benchmark	min.	CPU	Dec.	NCB	Jump	CPU	Dec.	NCB	Jump
cordic_Fa2@0	6	0.14	48	14	5	0.21	47	14	4
cordic_Fa3@0	6	0.16	53	14	5	0.18	52	14	4
cordic_Fa3@1	6	0.25	100	6	4	0.27	105	6	4
cordic_Fa4@1	6	0.17	84	4	3	0.18	84	4	3
misex1_Fd0@1	4	0.36	39	0	1	0.25	25	0	1
misex1_Fd1@0	4	0.32	53	3	5	0.18	38	4	5
misex1_Fd3@1	3	0.36	48	5	4	0.28	39	4	4
misex3_Fa@0	9	112.60	1352	34	7	52.02	834	78	14
misex3_Fa@1	9	42.09	756	25	5	30.58	642	56	9
misex3_Fb@0	9	313.87	1887	24	6	95.83	1152	119	9
misex3_Fb@1	8	96.27	1078	26	6	79.69	978	70	8
pcler8_Fi@1	2	0.41	78	9	2	0.39	78	9	2
pcler8_Fj@1	4	0.21	87	11	2	0.21	87	11	2
pcler8_Fk@1	4	0.53	119	3	2	0.51	121	4	2
term1_Fb@0	7	0.37	125	10	3	0.31	90	6	3
term1_Fb@1	7	0.31	140	13	3	0.27	100	7	3
term1_Fc@0	4	0.30	77	9	4	0.29	74	9	4
term1_Fd@1	4	0.39	92	10	9	0.32	85	10	8

Table 2: Lower bound explanations

constrained problem instances. Due to more adequate pruning techniques from SAT algorithms, *opbdp* and *min-prime* are able to solve more instances than the other algorithms, but *min-prime* shows to be more competitive than *opbdp* due to its greater search pruning ability. Despite the good results of these two algorithms, *bsolo* demonstrates to be the most competitive algorithm in this problem domain, mainly because *bsolo* is able to use features from both branch and bound and SAT algorithms.

As noted earlier, SAT-based BCP algorithms are better suited for instances whose constraints are hard to satisfy. In table 2 we present the results of *bsolo* for instances from the minimum-size test pattern problem [8]. This table includes the CPU times, the number of decisions, the number of non-chronological backtracks and the highest jump made in the search tree. On the left side of the table, *bsolo* was run without the lower bound explanation described in section 4 and the non-chronological backtracks are just due to logical or upper bound conflicts. On the right side, the lower bound explanation of section 4 is used and we can see that *bsolo* is able to increase the number of non-chronological backtracks while significantly reducing the amount of search and the execution time for most instances.

Finally, in table 3 we present a comparison between several algorithms for the set of instances from the minimum-size test pattern problem. Table 3 clearly shows that general purpose algorithms for solving 01-Integer Linear Programs (*lp-solve* and *cplex*) perform poorly. The same is true for *scherzo* which is not able to apply its main features to solving these instances. The SAT-based linear search algorithms *opbdp* [1] and *min-prime* [13] are able to solve all instances. Moreover, *bsolo* results are very competitive with the results of both *opbdp* and *min-prime*, mainly due to the new techniques proposed in this paper.

6 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. Besides detailing a branch-and-bound BCP algorithm built on top of a SAT solver, the paper describes conditions that allow for non-chronological backtracking in the presence of upper and lower bound conflicts. In addition, the paper also describes how reduction techniques, commonly used in BCP solvers, can be re-defined and utilized within a conflict analysis procedure, in such a way that non-chronological backtracking is enabled. 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 upper and lower bound conditions.

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.

A key aspect of the proposed techniques is the identification of a small set of dependencies explaining each identified conflict. In each case the main goal is to minimize the size of this set of dependencies, while guaranteeing that the resulting set still provides a sufficient explanation for the given conflict to occur. Future research work will naturally include seeking further simplification of the clauses created for each type of conflict. Moreover, additional techniques

		lp-solve	**scherzo**	**cplex**	**opbdp**	**min-prime**	**bsolo**
Benchmark	min.	CPU	CPU	CPU	CPU	CPU	CPU
cordic_Fa2@0	6	200.3	64.02	2.77	2.78	0.09	0.21
cordic_Fa3@0	6	969.5	67.84	2.20	5.99	0.09	0.18
cordic_Fa3@1	6	ub 7	97.37	9.02	3.95	0.83	0.27
cordic_Fa4@1	6	time	84.13	3.12	2.27	0.63	0.18
misex1_Fd0@1	4	261.7	0.39	59.47	0.06	0.14	0.25
misex1_Fd1@0	4	60.7	0.47	149.73	0.06	0.19	0.18
misex1_Fd3@1	3	24.0	0.26	72.07	0.06	0.14	0.28
misex3_Fa@0	9	time	mem.	time	180.42	85.70	52.02
misex3_Fa@1	9	time	mem.	time	207.81	111.24	30.58
misex3_Fb@0	9	time	mem.	time	1354.45	549.04	95.83
misex3_Fb@1	8	time	mem.	time	987.35	394.73	79.69
pcler8_Fi@1	2	19.8	2.37	3.52	0.74	0.40	0.39
pcler8_Fj@1	4	9.3	0.39	1.10	0.32	0.18	0.21
pcler8_Fk@1	4	8.2	0.28	5.48	0.48	0.28	0.51
term1_Fb@0	7	513.2	mem.	27.63	4.29	1.10	0.31
term1_Fb@1	7	404.6	256.42	22.83	8.85	0.62	0.27
term1_Fc@0	4	75.4	0.86	9.95	38.59	1.67	0.29
term1_Fd@1	4	150.3	1.50	11.82	12.14	1.93	0.32

Table 3: Algorithm comparison

from the SAT domain can potentially be applied to solving BCP. These techniques are likely to be significant 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] O. Coudert. Two-Level Logic Minimization, An Overview. *Integration, The VLSI Journal*, vol. 17(2):677–691, October 1993.

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

[5] O. Coudert and J. C. Madre. New Ideas for Solving Covering Problems. In *Proceedings of the ACM/IEEE Design Automation Conference*, June 1995.

[6] M.Ã. Tricks D. S. Johnson. Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1994.

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

[8] P. F. Flores, H. C. Neto, and J. P. Marques 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.

[9] J. Gimpel. A Reduction Technique for Prime Implicant Tables. *IEEE Transactions on Electronic Computers*, vol. EC-14:535–541, August 1965.

[10] E. Goldberg, L. Carloni, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Negative thinking by incremental problem solving: application to unate covering. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, 1997.

[11] G. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Pub., 1996.

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

[13] V. M. Manquinho, P. F. Flores, J. P. Marques 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.

[14] D. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

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

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

[17] J. P. Marques 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.

[18] 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.

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