Search Pruning Techniques in SAT-Based Branch-and-Bound Algorithms for the Binate Covering Problem

Vasco M. Manquinho and João P. Marques-Silva

***Abstract*—Covering problems are widely used as a modeling tool in electronic design automation. Recent years have seen dramatic improvements in algorithms for the unate/binate covering problem (UCP/BCP). Despite these improvements, BCP is a well-known computationally hard problem with many existing real-world instances that currently are hard or even impossible to solve. In this paper we apply search pruning techniques from the Boolean satisfiability domain to branch-and-bound algorithms for BCP. Furthermore, we generalize these techniques, in particular the ability to infer and record new constraints from conflicts and the ability to backtrack nonchronologically, to situations where the branch-and-bound BCP algorithm backtracks due to bounding conditions.**

Experimental results, obtained on representative real-world instances of the UCP/BCP, indicate that the proposed techniques are effective and can provide significant performance gains for specific classes of instances.

***Index Terms*—Backtrack search, binate covering problem, branch-and-bound, nonchronological backtracking, propositional satisfiability.**

I. Introduction

THE BINATE covering problem (BCP) finds many applications in electronic design automation (EDA) [10], [13], examples of which include logic and sequential synthesis (state minimization and exact encoding), cell-library binding, and minimization of Boolean relations [13]. 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. Examples of these techniques include, among others, partitioning [4], limit-lower bound [5], negative-thinking [9] (for unate covering), and linear-programming lower bounds [11]. Despite these improvements, and as with other NP-hard problems, new effective techniques may allow significant gains, both in the amount of search and in the run times. Besides efficiency improvements in solving existing problem instances, the ultimate benefit of devising new effective algorithmic techniques is the ability to solve new classes of problem 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], [15], [18]. In particular, and to our best knowledge, we provide for the first time conditions which enable branch-and-bound algorithms to backtrack *nonchronologically* whenever upper and lower bound conditions require bounding to take place. Moreover, we illustrate how value probing techniques can also be utilized in BCP solvers. One additional contribution of this paper is detailing the procedures for applying problem reduction techniques from the BCP domain to backtrack search algorithms. The proposed contributions allow the tight integration of BCP and SAT techniques within a unified algorithm for BCP.

This paper is organized as follows. In Section II 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 Sections IV and V 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. These include clause recording, nonchronological backtracking search strategies, and selective probing of variable assignments. Experimental results are presented in Section VI, and the paper concludes in Section VII.

II. Definitions

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

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

where c_j is a nonnegative 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

Manuscript received January 18, 2000; revised October 1, 2001. This work was supported in part by Fundação para a Ciência e Tecnologia (FCT) under Projects 1597/95, 11249/1998, 11266/1998, and 34504/1999. This paper was recommended by Associate Editor R. Gupta.

V. M. Manquinho is with the Computer Science Department, IST/Technical University of Lisbon, 1000-029 Lisbon, Portugal (e-mail: vasco.manquinho@inesc.pt).

J. P. Marques-Silva is with the Computer Science Department, IST/Technical University of Lisbon. He is also with INESC-ID and with Cadence European Labs, Lisbon, 1000-029 Lisbon, Portugal (e-mail: jpms@sat.inesc.pt).

Publisher Item Identifier S 0278-0070(02)02848-8.

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 utilization 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 $\overline{x}_j$. 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.

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 $\omega = (x_1 + \overline{x}_2)$ 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* [15].[1] 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 references (see for example [1] and [15]).

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, $\overline{x}_j$ can be represented by $1 - x_j$. For instance, the set of clauses $(x_1 + x_2 + x_3)$, $(\overline{x}_2 + \overline{x}_4)$, $(x_1 + \overline{x}_3)$ is equivalent of having the inequalities

$$\begin{aligned} x_1 + x_2 + x_3 &\geq 1 \\ -x_2 - x_4 &\geq -1 \\ x_1 - x_3 &\geq 0. \end{aligned}$$

These constraints could also be represented in a matrix like

$$\begin{bmatrix} 1 & 1 & 1 & \\ & -1 & & -1 \\ 1 & & -1 & \end{bmatrix} \geq \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}.$$

Notice that this definition fully complies with (1) for BCP, e.g., $b_2 = 1 - 2 = -1$.

Covering problems are often solved by branch-and-bound algorithms [4], [9], [16]. 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.

[1]In the UCP/BCP literature the repeated application of the unit clause rule corresponds to the identification of essential columns [3].

III. Backtrack Search Algorithms for Covering Problems

The most widely known approach for solving covering problems is the classical branch-and-bound procedure [10] that minimizes a cost function, 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. Each time a new lower cost solution is found, the upper bound value is updated. 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], [11], and [16] follow this approach.

There are several lower bound estimation procedures that can be used, namely the ones based on linear-programming relaxations [11] or Lagrangian relaxations [14], 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 III-D.

Covering algorithms also incorporate several powerful reduction techniques such as clause and variable dominance, row consensus, Gimpel's reduction [8], the limit lower bound theorem [5], and partitions [4]. A comprehensive overview of these methods can be found in [3] and [16].

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, incorporate different pruning strategies which are not commonly used in branch-and-bound algorithms for solving BCP. Moreover, in Section III-B an algorithm which combines features from both approaches is described.

A. SAT-Based Linear Search Algorithm

In [1], 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 [6] procedure, which is well known not to be competitive with modern state-of-the-art SAT solvers for the most representative real-world SAT problem instances. In [12], a new algorithm based on the GRASP SAT algorithm [15] is proposed, which is able to obtain better experimental results. Both these two algorithms interpret each instance of the binate covering problem

```
int min_prime(φ) {
  ub = ∑ c_j;
  while (ub ≥ 0) {
    φ = φ ∪ {∑ c_j · x_j < ub};
    status = solve_sat(φ);
    φ = φ − {∑ c_j · x_j < ub};
    if (status == SATISFIABLE)
      ub = ∑ c_j · x_j;
    else break;
  }
  return ub;
}
```

Fig. 1. SAT-based linear search algorithm.

as an instance of the SAT problem defined by the constraints $A \cdot x \geq b$, but with the additional constraint of having to find a solution with cost lower 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 defined to be

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

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 cost lower than the best (lowest) cost found so far.
2) Solve the resulting instance of the satisfiability problem, defined on linear inequalities. Adapting most SAT algorithms to deal with this generalization is straightforward [1].
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 instance does not have a solution.

B. SAT-Based Branch-and-Bound Algorithm

Additional SAT-based BCP algorithms have been proposed. In [12] 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. The *bsolo* algorithm incorporates the most significant features from both approaches, namely the bounding procedure and reduction techniques from branch-and-bound algorithms and the search pruning techniques from SAT algorithms.

```
int bsolo(φ) {
  ub = ∑ c_j + 1;
  while (TRUE) {
    if (!reduce_problem())
      return ub;
    identify_partitions();
    decide();
    if (!consistent_state())
      return ub;
    while (Estimate_LB() ≥ 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;
}
```

Fig. 2. SAT-based branch-and-bound algorithm.

Originally, the *bsolo* algorithm presented in [12] already incorporated the main pruning techniques of the GRASP SAT algorithm [15]. To our knowledge, *bsolo* was the first branch-and-bound algorithm for solving BCP that implemented a nonchronological backtracking search strategy, clause recording, and identification of necessary assignments. Mainly due to an effective conflict analysis procedure which allows nonchronological backtracking steps to be identified, *bsolo* performs better than other branch-and-bound algorithms in specific classes of instances, as shown in [12]. However, nonchronological backtracking was limited to one specific type of conflict, i.e., logical conflicts.[2] In Section IV we describe how to apply nonchronological backtracking also to other types of conflicts. The main steps of the algorithm (see Fig. 2) can be described 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,

[2]See Section III-C for a full description of all types of conflicts.

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 which decision node to backtrack to (using function *consistent_state*). Continue the search from Step 2).

C. Bound Conflicts

In *bsolo* two types of conflicts 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 nonchronological 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 [15]. 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 $\overline{x}_j$ (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. 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 $\omega_{\mathrm{bc}} = (\overline{x}_1 + x_2 + x_3 + \overline{x}_4)$. Again, the drawback of this approach (which was used in [12]) is that backtracking due to bound conflicts is always chronological, since it depends on all decision assignments made. In Section IV, we propose a new procedure to build these clauses, which enables nonchronological backtracking due to bound conflicts.

```
maximal_independent_set(φ) {
  MIS = ∅;
  do {
    ω = choose_clause(φ);
    MIS = MIS ∪ {ω};
    φ = delete_intersecting_clauses(φ,ω);
  } while (φ ≠ ∅);
  return MIS;
}
```

Fig. 3. Algorithm for computing a *MIS*.

D. Maximum Independent Set of Clauses

The maximum independent set of clauses (*MIS*) is a greedy method to estimate a lower bound on the value of the cost function based on an independent set of clauses. (A more detailed definition can be found for example in [3].)

The greedy procedure consists of finding a set *MIS* of disjoint unate clauses, i.e., clauses with only positive literals and with no literals in common among them. Since maximizing the cost of *MIS* is an NP-hard problem, a greedy computation is used, as shown in Fig. 3. The lower bound returned by this method can be arbitrarily far from the optimum and its effectiveness largely depends on the clauses included in *MIS*. Usually, one chooses the clause which maximizes the ratio between its weight and its number of elements.

The minimum cost for satisfying *MIS* is a *lower bound* on the solution of the problem instance and is given by

$$\mathrm{Cost}(MIS) = \sum_{\omega \in MIS} \mathrm{Weight}(\omega) \tag{4}$$

where

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

IV. SAT-Based Pruning Techniques for BCP

One of the main features of *bsolo* is the ability to backtrack nonchronologically when conflicts occur. This feature is enabled by the conflict analysis procedure inherited from the GRASP SAT algorithm. However, as illustrated in Section III-C, in the original *bsolo* algorithm nonchronological 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 nonchronological backtracking can take place due to bound conflicts.

A. Dependencies in Bound Conflicts

A bound conflict in an instance of the BCP C arises when the lower bound is equal to or higher than the upper bound. This condition can be written as $C.\mathrm{path} + C.\mathrm{lower} \geq C.\mathrm{upper}$, where $C.\mathrm{path}$ is the cost of the assignments already made, $C.\mathrm{lower}$ 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 $C.\mathrm{upper}$ is the best solution found so far. From the previous equation, we can readily conclude that $C.\mathrm{path}$ and $C.\mathrm{lower}$ are the unique components involved in each bound conflict. (Notice that $C.\mathrm{upper}$ is just the lowest value of the cost function for all assignments satisfying the constraints that have been computed earlier in the search process.) Therefore, we will analyze both $C.\mathrm{path}$ and $C.\mathrm{lower}$ components in order to establish the assignments responsible for a given bound conflict.

We start by studying $C.\mathrm{path}$. Clearly, the variable assignments that cause the value of $C.\mathrm{path}$ to grow are solely those assignments with a value of 1. Hence, we can define a set of literals for the current search path ω_{cp}, such that each variable in ω_{cp} has positive cost and is assigned value 1. This condition is stated as follows:

$$\omega_{\mathrm{cp}} = \{l = \overline{x}_j \colon \mathrm{Cost}(x_j) > 0 \wedge x_j = 1\} \tag{6}$$

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

We now consider $C.\mathrm{lower}$. Let *MIS* be the independent set of clauses, obtained by the method described in Section III-D, that determines the value of $C.\mathrm{lower}$. Observe that each clause in *MIS* is part of *MIS* because it is neither satisfied nor has common literals with any other clause in *MIS*. Clearly, for each clause *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 $C.\mathrm{lower}$

$$\omega_{cl} = \{l \colon l = 0 \wedge l \in \omega_i \wedge \omega_i \in MIS\}. \tag{7}$$

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

$$\omega_{\mathrm{bc}} = \omega_{\mathrm{cp}} \cup \omega_{cl}. \tag{8}$$

(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 $C.\mathrm{path}$ and $C.\mathrm{lower}$ 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 [15]. We should observe that backtracking can be nonchronological because clause ω_{bc} does not necessarily depend on all decision assignments. Moreover, the clause recording mechanism from GRASP allows ω_{bc} to 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 decision assignments is never repeated in the search process.

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

In order to illustrate a bound conflict situation, consider the following example.[3] Suppose we have $\omega_1 = (x_1 + x_4)$, $\omega_2 = (x_3 + x_4)$, $\omega_3 = (\overline{x}_2 + \overline{x}_4)$ and $\omega_4 = (\overline{x}_3 + x_4)$ in our problem formulation where $x_1 + x_2 + x_3$ defines the objective function. Suppose also that some decision variables assignments are made, namely $x_1 = 1$ in decision level 1, $x_2 = 0$ in decision level 2, and $x_3 = 1$ in decision level 3. Therefore, x_4 must be assigned value 1, a solution is found for the problem, and we have $C.\mathrm{upper} = 2$. A bound conflict is then issued and in order to solve this conflict we must have either $x_1 = 0$ or $x_3 = 0$. From (8) we build the bound conflict clause $(\overline{x}_1 + \overline{x}_3)$. After backtracking and undo the decision variable assignment $x_3 = 1$, from the bound conflict clause we must have $x_3 = 0$. Again, x_4 must be assigned value 1 and a new solution is found. Now we have $C.\mathrm{upper} = 1$ because of the assignment of 1 to x_1. From (8) we build a new bound conflict clause $(\overline{x}_1)$ and the search process can backtrack nonchronologically to decision level 1. Notice that the decision assignment in level 2 is considered irrelevant and the search is pruned at that decision level.

B. Reducing Dependencies in Bound Conflicts

With respect to (8) a more careful analysis allows us to conclude that not all literals in ω_{bc} are actually necessary. Suppose that the lower bound estimation is higher than the upper bound and define this difference as $\mathrm{diff} = (C.\mathrm{path} + C.\mathrm{lower}) - C.\mathrm{upper}$. It is clearly true that if $C.\mathrm{path}$ was decreased by diff, the bound conflict would still hold since we would then have $C.\mathrm{upper} = C.\mathrm{path} + C.\mathrm{lower}$. Therefore, we may conclude that not all assignments in $C.\mathrm{path}$ are necessary to explain 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 total 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 nonchronologically 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 $C.\mathrm{lower}$, then we must have $l \in \omega_{\mathrm{bc}}$ and there is no reduction in the dependencies of the conflict clause ω_{bc}.

[3]This example is necessarily small and solely intended to illustrate the main points.

Moreover, it is also interesting to observe that a clause resulting from a bound conflict can be simpler. We have described how simplifications can be made to the $C.\mathrm{path}$ component, but other simplifications can also be applied to the literals added due to the independent set of clauses (*MIS*), i.e., ω_{cl}. 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_{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 $C.\mathrm{lower}$ component would be lower.[4] However, since the cost of the variable is higher than or equal to the minimum cost of ω_i, the $C.\mathrm{path}$ 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}.

C. Applying Dependency Reduction Techniques

For a better understanding of the techniques mentioned in the paper, we will present an example on how a conflict clause can be built and the application of the dependency reduction techniques is effective.

Consider that at some point of the search process we have $C.\mathrm{path} = 3$ from the set of assignments $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, $x_4 = 1$, and $x_5 = 0$ where all problem variables have a cost of 1 in the cost function. Consider also that we have $C.\mathrm{lower} = 3$ from the independent set of clauses $\omega_1 = (x_2 + \overline{x}_3 + x_8 + x_9)$, $\omega_2 = (\overline{x}_3 + x_5 + x_6 + x_7)$, and $\omega_3 = (\overline{x}_4 + x_5 + x_{10} + x_{11})$ where variables x_6, x_7, x_8, x_9, x_{10}, and x_{11} are unassigned.

Suppose the best solution found so far has a cost of 5 ($C.\mathrm{upper} = 5$). Hence, a bound conflict situation has been identified, since $C.\mathrm{path} + C.\mathrm{lower} \geq C.\mathrm{upper}$, and the search can be bound. As described in Section IV-A, in order to bound the search, our algorithm will add an unsatisfied clause explaining the conflict. Afterwards, the GRASP conflict analysis procedure will be carried out to determine to which level of the search tree can the algorithm backtrack without loosing completeness.

The conflict explanation clause is created as proposed in Section IV-A. From (6) we have $\omega_{\mathrm{cp}} = (\overline{x}_1 + \overline{x}_3 + \overline{x}_4)$ and from (7) we have $\omega_{cl} = (x_2 + \overline{x}_3 + \overline{x}_4 + x_5)$. Therefore, the bound conflict explanation clause can be built as proposed in (8) and we have $\omega_{\mathrm{bc}} = (\overline{x}_1 + x_2 + \overline{x}_3 + \overline{x}_4 + x_5)$. The bound conflict clauses implicity state which variables should be unassigned or have a different value to proceed the search. In this example, either x_1, x_3, or x_4 should be assigned value 0 or x_2 or x_5 be assigned value 1.

In Section IV-B we presented some techniques to reduce dependencies in conflict clauses from bound conflicts. The application of such techniques is important since with a smaller set of dependencies it is more likely that a nonchronological backtrack step can occur.

Notice that in our small example, the lower bound estimation is higher than the best solution found so far and we have $\mathrm{diff} = 1$ as defined in Section IV-B. In these conditions, a greedy procedure can be applied to choose which literal to remove from ω_{cp}. For instance, if we remove x_1 from ω_{cp}, ω_{cp} no longer justifies $C.\mathrm{path} = 3$, but it is sufficient to justify $C.\mathrm{path} = 2$. Notice that with $C.\mathrm{path} = 2$ the bound conflict still holds and, therefore, $\omega_{\mathrm{bc}} = (x_2 + \overline{x}_3 + \overline{x}_4 + x_5)$ is enough to explain the bound conflict.

No more reductions can be made due to diff, since now we have $\mathrm{diff} = 0$, but reductions can be made on ω_{cl}. Notice that x_2 only appears in one clause of the independent set and is assigned value 0. If x_2 was instead assigned value 1, the conflict would still hold since $C.\mathrm{path}$ would be higher. The value of x_2 is irrelevant for the conflict situation and can be removed from ω_{bc}. Therefore, we have $\omega_{\mathrm{bc}} = (\overline{x}_3 + \overline{x}_4 + x_5)$ as the bound conflict clause for this example.

D. More on Dependencies in Bound Conflicts

As we have shown in Section IV-A, whenever a bound conflict occurs, it is necessary to establish which assignments explain the conflict. The main purpose for doing so is that the conflict may not depend on the most recent decision assignments and, consequently, nonchronological backtracking can occur.

Finding a set of decision assignments which explain the conflict is straightforward, but if the size of the explanation can be reduced, it is more likely that the consequent backtrack step be nonchronological. Therefore, it is of key importance to find a small set of assignments that explains each bound conflict. In the previous section we showed how this set of assignments can be identified and also proposed some simplifications which might be applied to reduce the size of the explanation. This section illustrates how a more careful analysis can introduce additional simplifications, allowing the elimination of further assignments from the conflict explanation.

As illustrated in Section IV-A, the number of dependencies from $C.\mathrm{path}$ in bound conflicts can be reduced whenever $\mathrm{diff} > 0$, where $\mathrm{diff} = (C.\mathrm{path} + C.\mathrm{lower}) - C.\mathrm{upper}$. However, the same principle can be applied to dependencies from $C.\mathrm{lower}$. Notice that if we remove a subset of clauses D_MIS from *MIS* (used to obtain $C.\mathrm{lower}$) such that

$$\mathrm{Cost}(D\_MIS) \leq \mathrm{diff} \tag{9}$$

where

$$\mathrm{Cost}(D\_MIS) = \sum_{\omega \in D\_MIS} \mathrm{Weight}(\omega) \tag{10}$$

then the bound conflict will still hold since $C.\mathrm{upper} \leq C.\mathrm{path} + C.\mathrm{lower}$, but $C.\mathrm{lower}$ is now obtained from the independent set of clauses $MIS \setminus D\_MIS$. Therefore, the bound conflict clause ω_{bc} can still be built using (8), but the ω_{cl} can now be reformulated as

$$\omega_{cl} = \{l\colon l = 0 \wedge l \in \omega_i \wedge \omega_i \in MIS \setminus D\_MIS\}. \tag{11}$$

Moreover, the simplifications described in Section IV-A can also be applied to the resulting ω_{cl}.

One should note that the reduction on the number of dependencies depends 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},

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

```
reduce_independent_set(MIS, diff ) {
  D_MIS = empty set;
  do{
    ω = choose_MIS_clause(MIS, diff);
    D_MIS = D_MIS ∪ {ω};
    MIS = MIS \ ω;
    diff --;
  } while ((MIS not empty) ∧ (diff > 0));
}
```

Fig. 4. Algorithm for reducing *MIS*.

```
reduce_path_dependencies(MIS, ω_cp) {
  NEW_MIS = MIS;
  while (ω = choose_SAT_clause(NEW_MIS, ω_cp)) {
    NEW_MIS = NEW_MIS ∪ ω;
    sat_l = sat_literal(ω);
    ω_cp = ω_cp \ sat_l;
  }
}
```

Fig. 5. Algorithm for reducing $C.\mathrm{path}$ dependencies.

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. As an example of reducing ω_{bc}, in the Appendix we propose a model for computing a minimum number of dependencies in ω_{cl}, given a fixed ω_{cp}.

In order to get a small set of dependencies, *bsolo* has a greedy procedure which heuristically approximates the best D_MIS that would reduce the number of dependencies to a minimum. We know from (7) that if a clause ω_i from *MIS* has an unsatisfied literal l_j, then l_j will be in ω_{cl}. If we can remove at most diff clauses from *MIS*, the literals which occur in more clauses of MIS than the value of diff will always be in some clause in *MIS*, and from (7), l_j must be in ω_{cl}. Therefore, the clauses to be removed from *MIS* are the ones that maximize the number of literals that belong to fewer clauses in *MIS* than the value of diff, i.e., the number of literals that can still be removed from ω_{cl} due to this procedure. Fig. 4 outlines this procedure. At each step, the clause which contains more literals with potential to be eliminated from ω_{cl} is selected. If after this reduction in ω_{cl} we still have $\mathrm{diff} > 0$, the reduction of ω_{cp} due to diff described in Section IV-A can also be applied.

So far we have presented several procedures to reduce the number of dependencies in bound conflicts. Among others, it was explained how to reduce dependencies when the lower bound value is higher than the upper bound ($\mathrm{diff} > 0$). After the application of such reduction procedures, we usually have $\mathrm{diff} = 0$ and, therefore, we have a set ω_{bc} which explains a bound conflict on the form $C.\mathrm{upper} = C.\mathrm{path} + C.\mathrm{lower}$.[5]

Let l_j be a literal such that $l_j \in \omega_{\mathrm{cp}}$ and $l_j \notin \omega_{cl}$. Then l_j is in ω_{bc} only due to the $C.\mathrm{path}$ component of the bound conflict. Let *MIS* be the independent set, computed with the procedure described in Fig. 5, which is used to obtain the value of $C.\mathrm{lower}$. If there is a satisfied clause ω_i such that $\overline{l}_j$ is the only literal which currently satisfies ω_i, then l_j can be removed from ω_{cp} under certain conditions. Namely, all other literals of ω_i must be positive, unassigned, and must not intersect *MIS* (so that ω_i can be added to *MIS*). Moreover, all of them must have a cost higher than or equal to l_j and no clause in *MIS* can contain l_j.

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

Suppose that variables x_1, x_2, and x_3 belong to the cost function and $x_1 = 1$. If a bound conflict occurs, from (6) $\overline{x}_1$ 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*, $\overline{x_1}$ 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.

E. Handling Reduction Techniques

As mentioned in the previous sections, for implementing nonchronological 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 nonchronologically. 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 [4]. For the other reduction techniques, a similar approach is used.

The limit lower bound theorem is applied to a variable x_j whenever

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

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 $C.\mathrm{path}$ and $C.\mathrm{lower}$. 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:

$$\omega_{llb} = \omega_{\mathrm{cp}} \cup \omega_{cl} \cup \{\overline{x}_j\} \qquad (13)$$

where ω_{cp} and ω_{cl} are the literals which explain the values in $C.\mathrm{path}$ and $C.\mathrm{lower}$, as described in Section IV-A. 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}.) Moreover, clause ω_{llb} can also be used later on in the search process to imply necessary assignments if its

[5] In (8), ω_{bc} explained a conflict on the form $C.\mathrm{upper} \leq C.\mathrm{path} + C.\mathrm{lower}$.

state becomes unit. In those cases, the limit lower bound is applied automatically during the Boolean constraint propagation phase (see Section II).

V. Probing Variable Assignments

The decisions made during the search process are vital for the efficiency of the algorithm. With this in mind, in this section we propose a new strategy that anticipates whether a decision leads to a conflict. The process of probing variable assignments consists of testing the assignment of 0 or 1 to unassigned variables and, if a conflict is reached while testing an assignment v to a variable x_j, then the opposite value $\overline{v}$ is implied for variable x_j.

Probing the assignment of a Boolean value v to a variable x_j consists of analyzing the result of Boolean constraint propagation in case the assignment is made. In cases where no conflict (logic or due to lower bound) is detected, the opposite value of v, $\overline{v}$, is assigned to x_j and the same analysis is performed. In either case, whenever a conflict is reached the opposite value of the assignment that led to the conflict situation is automatically implied. This procedure is referred to as *complete probing of variable assignments*.

However, this probing process involves significant computational overhead, mainly due to the application of Boolean constraint propagation and lower bound computation. An alternative approach is to use instead *restricted probing of variable assignments*. In restricted probing, when we test the assignment of value v to a variable x_j, our main goal is to simply check whether the assignment results in an immediate increase of the lower bound value (by increasing $C.\mathrm{path}$). Hence, instead of performing complete Boolean constraint propagation, we only check the binary clauses (with just two free literals) that contain variable x_j. By assigning x_j, these clauses either become satisfied or unit, in which case new assignments will be implied. However, in restricted probing, Boolean constraint propagation is not carried out any further. Instead, we check whether these deduced assignments increase the value of the lower bound. If the lower bound becomes equal to or higher than the upper bound, then the complemented value of v, $\overline{v}$, can be implied for x_j.

Suppose that at a certain point of the search process, we have $C.\mathrm{upper} = 5$, $C.\mathrm{lower} = 2$, and $C.\mathrm{path} = 1$ and all variables assigned value 1 would add just 1 to the cost function. Notice that the limit lower bound theorem cannot be applied since $C.\mathrm{upper} - C.\mathrm{lower} - C.\mathrm{path} = 2$, which is higher than the cost of every variable in the cost function. Suppose we have (among others) the following set of unresolved clauses still to satisfy:

$$\omega_1 = (\overline{x}_1 + x_2);$$
$$\omega_2 = (\overline{x}_1 + x_3);$$
$$\omega_3 = (x_3 + x_7 + x_8);$$
$$\omega_4 = (x_5 + x_6);$$

and our approximation of the maximum independent set which is used to estimate $C.\mathrm{lower}$ is:

$$\omega_3 = (x_3 + x_7 + x_8);$$
$$\omega_4 = (x_5 + x_6)..$$

Suppose that we would test the assignment $x_1 = 1$. In this case, just by checking the binary clauses (ω_1 and ω_2) we can conclude that $x_2 = 1$ and $x_3 = 1$ are necessary assignments. Therefore, if we make the assignment $x_1 = 1$, we would have $C.\mathrm{lower} = 1$ (since ω_3 would become satisfied) and $C.\mathrm{path} = 4$, resulting in a lower bound conflict. Since the assignment $x_1 = 1$ would result in a conflicting condition, we know that $x_1 = 0$ is a necessary assignment due to the application of restrictive probing on the assignment $x_1 = 1$.

Restricted probing of variable assignments can be formally described as follows. Let us consider the assignment of value v to x_j and let $V(MIS)$ be the set of unassigned variables in the independent set of clauses *MIS* used to compute $C.\mathrm{lower}$. Let V_0 and V_1 be two variable sets defined as follows:

$$\begin{aligned} V_0 = \{x_i\colon\ & \mathrm{unresolved}((x_i + x_j)) \wedge \mathrm{Cost}(x_i) \\ & > 0 \wedge x_i \notin V(MIS)\} \\ V_1 = \{x_i\colon\ & \mathrm{unresolved}((x_i + \overline{x}_j)) \wedge \mathrm{Cost}(x_i) \\ & > 0 \wedge x_i \notin V(MIS)\}. \end{aligned} \tag{14}$$

V_0 and V_1 define the sets of the variables which are immediately implied value 1 whenever $x_j = 0$ and $x_j = 1$, respectively. Moreover, these variables do not belong to any clause in *MIS* and have positive cost.[6] Consequently, these are the variables that will increase the lower bound if an assignment to x_j is made. Clearly, the cost associated with set V_v is the sum of the costs of the variables in V_v.

Let $\tau(x_j, v)$ be a function such that

$$\tau(x_j, v) = \begin{cases} \text{if } v = 0 \vee x_j \in MIS \\ \mathrm{Cost}(x_j) \quad \text{otherwise.} \end{cases} \tag{15}$$

Therefore, if the condition

$$C.\mathrm{path} + C.\mathrm{lower} + \mathrm{Cost}(V_v) + \tau(x_j, v) \geq C.\mathrm{upper} \tag{16}$$

is true, then the complemented value of v, $\overline{v}$ is a necessary assignment for x_j.

Notice that the limit lower bound theorem [5] can be interpreted as a particular case of probing variable assignments. The limit lower theorem is applied to a variable x_j when $C.\mathrm{upper} - (C.\mathrm{path} + C.\mathrm{lower}) \leq \mathrm{Cost}(x_j)$ and can only imply the value 0 for x_j. The process of probing variable assignments is able to imply both value 0 or 1 and can be applied even when $C.\mathrm{upper} - (C.\mathrm{path} + C.\mathrm{lower}) > \mathrm{Cost}(x_j)$. Moreover, probing can also be applied to variables that are not in the cost function. We should observe, however, that this procedure is computationally less efficient than the limit lower bound when applied to the same cases.

Section IV-E explains why a new clause must be added when the limit lower bound theorem is applied. Furthermore, it also described how this clause should be built. In probing variable assignments, as in any other problem reduction technique, the same must be done and a new clause must be built. However, for each probing reduction technique a different set of explanations must be considered.

Clearly, restricted probing depends on the lower bound value. Consequently, the explanations for the value of $C.\mathrm{path}$ and $C.\mathrm{lower}$ must be present in the new clause. Moreover, we

[6]Observe that x_i can be in $V(MIS)$ as long as enough variables in $V(MIS)$ are picked and cause the decrease in the value of the lower bound to be offset by the increase in the path value.

also have to consider the clauses in V_v from (14) used during probing. In addition to $C.\text{path}$ and $C.\text{lower}$, the clauses in V_v are also necessary for probing to yield necessary assignments. Consequently, we must also identify the set of assignments responsible for these clauses to be in V_v. An explanation for this fact is the set of literals assigned value 0 in those clauses. Let ω_{V_v} be the set of literals assigned value 0 in the clauses of V_v that are considered while probing the assignment $x_j = v$. Thus, the clause ω_{pro} explains the implied variable assignment obtained by applying probing and can be defined as follows:

$$\omega_{pro} = \begin{cases} \omega_{\text{cp}} \cup \omega_{cl} \cup \omega_{V_1} \cup \{\overline{x}_j\} \\ \quad \text{to explain the assignment } x_j = 0 \\ \omega_{\text{cp}} \cup \omega_{cl} \cup \omega_{V_0} \cup \{x_j\} \\ \quad \text{to explain the assignment } x_j = 1 \end{cases} \tag{17}$$

where ω_{cp} and ω_{cl} are defined as in Section IV. Notice that when created ω_{pro} is unit, and so it implies the value of x_j as intended.

VI. Experimental Results

In this section, we include experimental results of several algorithms in two different sets of benchmarks. Table I presents results for instances of the MCNC benchmark suite [17], whereas the remaining tables present results for instances of the minimum-size test pattern problem [7].

For the experimental results given below, the CPU times were obtained on a SUN Sparc Ultra I, running at 170 MHz, 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 h. When the algorithm was unable to solve the instance due to time restrictions, the best upper bound found at the time is shown. Otherwise, if no upper bound was computed, the reason of failure is shown, which was either due to the time (`time`) or memory (`mem.`) limits imposed. In Table I, besides the time taken and the number of decisions made to solve the instances (`Dec.`), it is also shown the number of nonchronological backtracks (`NCB`) and the highest jump made in the search tree (`Jmp.`).

In Table I we present a comparison between *bsolo* and *scherzo* on the MCNC benchmark set.[7] *scherzo* is a classical branch-and-bound algorithm with powerful problem reduction techniques and very effective for this set of benchmarks, since most clauses are unate (i.e., only have positive literals). Clearly, *scherzo* is able to solve more instances and is, in general, faster. In this benchmark set, the main features of *bsolo* are not extensively used. We note, however, that there are some problem instances in which fewer decisions are made by *bsolo*.

In general, the bookkeeping associated with implementing the proposed SAT-based pruning techniques can introduce noticeable computational overhead in *bsolo*. For the instances above, the gains obtained from applying the SAT-based techniques are small since nonchronological backtracking is almost nonexisting, suggesting that further work must be done toward reducing the total number of dependencies.

As noted earlier, SAT-based BCP algorithms are better suited for instances whose constraints are hard to satisfy. In Table II we present the results of *bsolo* for instances from the minimum-size test pattern problem [7]. (As shown in [7], the minimum-size test pattern problem can be formulated as a special case of the binate covering problem.) Each problem instance captures the test pattern minimization problem in which the objective is to compute test patterns with a minimum number of specified primary input assignments. For example, `duke_Fv5@1` denotes the problem instance defining the minimum-size test pattern problem for circuit duke2 with fault given by line Fv5 stuck-at 1.

TABLE I
Results for *bsolo* and *scherzo*

		bsolo		scherzo	
Benchmark	min.	CPU	Dec.	CPU	Dec.
5xp1.b	12	11.42	1314	4.5	2234
9sym.b	5	10.78	263	3.6	320
alu4.b	–	ub 51	time	–	time
apex4.a	776	ub 792	time	87.4	48359
bench1.pi	–	ub 123	time	–	time
clip.b	15	6.52	1734	0.6	97
count.b	24	2.66	94	478.0	299780
e64.b	–	ub 48	time	–	mem.
ex5.pi	–	ub 67	time	–	time
exam.pi	–	ub 63	time	–	time
f51m.b	18	9.10	2766	1.9	1586
jac3	15	ub 17	time	4.9	292
max1024.pi	–	ub 262	time	–	time
prom2.pi	–	ub 305	time	–	time
rot.b	–	ub 121	time	–	time
sao2.b	25	1.32	444	0.9	279
test4.pi	–	ub 101	time	–	time

In this table, and besides the CPU time and the number of decisions, the number of nonchronological backtracks and the highest jump made in the search tree are also included. On the left side, *bsolo* does not use the bound explanation techniques described in Section IV and nonchronological backtracking is just due to logical conflicts [12]. On the right side, both the upper and lower bound explanation of Section IV and restricted probing of variable assignments are used. As we can readily conclude, for most instances *bsolo* is able to increase the number of nonchronological backtracks while significantly reducing the amount of search and the execution time.

Table II shows that the use of conflict explanations increases the number of nonchronological backtracks, thus proving that nonchronological backtracking can be observed in bound conflicts. We should note that in earlier versions of *bsolo* [12], only logical-based conflicts were able to produce nonchronological backtracks. Moreover, by using bound conflict explanations, *bsolo* reduces the number of explored nodes on the search tree, therefore improving its efficiency. In several cases we can observe an increase on both the number of nonchronological backtracks and on the highest jump in the search tree. For example, instance `c3540_F20@1` could not be solved with *bsolo* when not using explanations, but was solved in less than one third of the given time limit with the identification of dependencies in bound conflicts.

Finally, in Table III we present a comparison between several algorithms for this set of instances. Table III clearly shows that the general purpose algorithm for solving 01-Integer Linear Programs *lp-solve* performs poorly. The same is true for *scherzo*

[7]Results from other algorithms not shown in this table since they were unable to solve any of the instances in the given time limit.

TABLE II
LOWER BOUND EXPLANATIONS

bsolo		no explanations				using explanations			
Benchmark	min.	CPU	Dec.	NCB	Jmp.	CPU	Dec.	NCB	Jmp.
c1908_F469@0	11	2106.43	105181	19	8	1405.03	65563	592	41
c1908_F953@0	4	143.73	3036	3	3	105.02	2131	61	41
c3540_F20@1	6	ub6	38002	137	10	1112.60	10601	771	56
c432_F1gat@1	8	282.46	22983	31	5	52.11	4637	804	28
c432_F37gat@1	9	ub14	834819	15	4	1020.11	291450	58822	30
c499_Fic2@1	–	ub41	1000030	0	1	ub41	1000066	1	38
c6288_F35gat@1	4	187.13	2160	1	3	43.84	731	34	67
c6288_F69gat@1	6	1486.77	13052	129	5	312.20	3829	263	63
9symml_F1@1	9	2.52	309	5	2	2.64	300	23	17
9symml_F6@0	9	2.52	298	8	2	2.58	294	16	14
alu4_Fj@0	6	60.56	1244	25	4	49.46	1077	73	21
alu4_Fl@1	6	59.34	1042	14	4	36.92	681	41	14
apex2_Fv14@1	10	3.76	765	0	1	2.60	451	36	15
apex2_Fv17@1	12	4.43	924	1	4	2.64	447	43	12
duke2_Fv5@1	5	9.86	496	10	3	5.04	390	25	18
duke2_Fv7@0	5	4.93	383	0	1	3.62	342	18	24
misex3_Fa@0	9	28.09	1330	17	6	15.98	718	52	32
misex3_Fb@1	8	27.12	1025	5	5	23.96	873	43	42
spla_Fv10@0	7	16.25	689	2	3	11.61	647	16	47
spla_Fv14@0	8	16.82	889	2	2	9.83	693	32	66

TABLE III
ALGORITHM COMPARISON

		lp-solve	scherzo	opbdp	min-prime	bsolo
Benchmark	min.	CPU	CPU	CPU	CPU	CPU
c1908_F469@0	11	time	time	ub 24	ub 29	1405.03
c1908_F953@0	4	time	3424.81	ub 26	ub 15	105.02
c3540_F20@1	6	time	mem.	ub 13	2672.40	1112.60
c432_F1gat@1	8	ub 15	time	1148.27	901.90	52.11
c432_F37gat@1	9	time	time	3574.44	447.39	1020.11
c499_Fic2@1	–	time	time	ub 41	ub 41	ub41
c5315_F43@0	3	2.6	0.92	30.38	10.08	0.44
c5315_F54@1	5	time	mem.	time	ub 38	17.68
c6288_F35gat@1	4	time	mem.	1330.95	128.66	43.84
c6288_F69gat@1	6	time	mem.	ub 9	ub 7	312.20
9symml_F1@1	9	ub 9	28.64	3.15	8.13	2.64
9symml_F6@0	9	ub 9	29.44	1.43	17.99	2.58
alu4_Fj@0	6	time	879.05	413.71	29.42	49.46
alu4_Fl@1	6	time	1638.98	557.14	14.82	36.92
apex2_Fv14@1	10	ub 10	mem.	639.25	13.73	2.60
apex2_Fv17@1	12	time	mem.	545.97	48.89	2.64
duke2_Fv5@1	5	time	mem.	90.02	22.80	5.04
duke2_Fv7@0	5	time	mem.	24.83	6.50	3.62
misex3_Fa@0	9	time	mem.	180.42	85.70	15.98
misex3_Fb@1	8	time	mem.	987.35	394.73	23.96
spla_Fv10@0	7	time	mem.	202.98	33.61	11.61
spla_Fv14@0	8	time	mem.	264.23	64.84	9.83

which is not able to apply its main features in solving these instances. The SAT-based linear search algorithm *opbdp* [1] is able to solve most of the benchmarks. Similarly, we can observe that *min-prime* [12] can also solve most instances, with better results mainly due to the incorporation of the features from GRASP SAT algorithm [15]. Moreover, *bsolo* is in general faster than both *opbdp* and *min-prime*, mainly due to the new techniques proposed in this paper. One should note that in almost all cases where *bsolo* takes more time than *min-prime* to solve the problem instance, the number of decisions made by *bsolo* is smaller than the number of decisions made by *min-prime*. The time overhead of the features incorporated in *bsolo* which are not present in *min-prime* (namely problem reduction techniques, lower bound estimation, limit lower bound, probing, and explanation for bound conflicts, among others) are responsible for these results.

VII. Conclusion

This paper extends well-known search pruning techniques, from the Boolean satisfiability domain, to branch-and-bound algorithms for solving the unate and binate covering problems. Besides detailing a branch-and-bound BCP algorithm built on top of a SAT solver, the paper describes conditions that allow for nonchronological backtracking in the presence of bound conflicts. In addition, the paper also describes how reduction techniques, commonly used in BCP solvers, can be redefined and utilized within a conflict analysis procedure, in such a way that nonchronological backtracking is enabled. To our best knowledge, this is the first time that branch-and-bound algorithms are augmented with the ability for backtracking nonchronologically in the presence of conflicts that result from bound conditions. Moreover, we also describe simplification techniques for the explanations of bound conflicts. Finally, we have shown how probing techniques, also commonly used in the Boolean satisfiability domain, can be extended to algorithms for the binate covering problem.

Preliminary results obtained on several instances of the unate and binate covering problems 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 bound conflicts. Moreover, additional techniques 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.

Appendix
Minimizing Dependencies in Bound Conflicts

In this Appendix we derive an optimization model for computing a minimum number of dependencies for explaining the current value of *MIS*. Notice that we are not explicitly minimizing the size of $\omega_{\rm bc}$, but solely the size of ω_{cl} taking into account the set of literals in $\omega_{\rm cp}$, which is assumed to be *fixed*.

Without loss of generality, let l_i, with $i \in \{1, \ldots, k\}$, be the set of literals in clauses of *MIS* that have been assigned value 0. Furthermore, let ω_{i_j}, with $j \in \{1, \ldots, k_i\}$, denote each clause in *MIS* that contains literal l_i that is assigned value 0. We define variable y_i to be 1 if and only if literal l_i is included in the final set of dependencies explaining the value of *MIS* (i.e., ω_{cl}). In addition, we define z_{i_j} to be 1 if and only if ω_{i_j} is *not* included in the resulting reduced *MIS*.

Each literal l_i is only required to be included in the final set of dependencies provided at least one of the clauses containing l_i is also included in the final *MIS*. Thus we can say that selecting l_i to be in the final set of dependencies implies that some clause containing l_i is also selected to include the final *MIS*. Consequently, this constraint can be formulated as follows:

$$y_i \rightarrow \sum_{j=1}^{k_i} \overline{z_{i_j}}. \tag{18}$$

In addition, the number of clauses eliminated from *MIS* has to be *no greater* than diff. As a result, another constraint is

$$\sum_{\omega_s \in MIS} z_s \leq \text{diff}. \tag{19}$$

Moreover, our goal is to minimize the number of dependencies that are *not* in $\omega_{\rm cp}$, since these are known to be already included in $\omega_{\rm bc}$. Thus, any literal l_i[8] already included in $\omega_{\rm cp}$ must not be considered for reducing the total number of dependencies. This yields the additional set of constraints

$$y_i = 1, \qquad \text{if } l_i \in \omega_{\rm cp}. \tag{20}$$

Finally, the cost function associated with minimizing the number of dependencies from the lower bound estimate becomes

$$\text{minimize} \sum_{i=1}^{k} y_i. \tag{21}$$

Putting it all together, we get the overall BCP problem formulation

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{k} y_i \\ \text{subject to} \quad & \sum_{\omega_s \in MIS} z_s \leq \text{diff} \\ & y_i = 1, \qquad \text{if } l_i \in \omega_{\rm cp} \wedge i \in \{1, \ldots, k\} \\ & \overline{y_i} + \sum_{j=1}^{k_i} \overline{z_{i_j}}, \qquad \text{if } l_i \notin \omega_{\rm cp} \wedge i \in \{1, \ldots, k\}. \end{aligned} \tag{22}$$

Clearly, and in general, our goal is not to solve exactly the above BCP formulation, but only to obtain approximate heuristic solutions.

References

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

[2] R. Bayardo, Jr. and R. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," in *Proc. Nat. Conf. Artificial Intelligence*, 1997.

[3] O. Coudert, "Two-level logic minimization, An overview," *Integration, VLSI J.*, vol. 17, no. 2, pp. 677–691, Oct. 1993.

[4] ——, "On solving covering problems," in *Proc. ACM/IEEE Design Automation Conf.*, June 1996.

[5] O. Coudert and J. C. Madre, "New ideas for solving covering problems," in *Proc. ACM/IEEE Design Automation Conf.*, June 1995.

[6] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. Assn. Computing Machinery*, vol. 7, pp. 201–215, 1960.

[7] P. F. Flores, H. C. Neto, and J. P. Marques Silva, "An exact solution to the minimum-size test pattern problem," in *Proc. IEEE Int. Conf. Computer Design*, Oct. 1998, pp. 510–515.

[8] J. Gimpel, "A reduction technique for prime implicant tables," *IEEE Trans. Electron. Computers*, vol. EC-14, pp. 535–541, Aug. 1965.

[8]Observe that l_i must correspond to a variable $\overline{x_l}$, where x_l is currently assigned value 1.

[9] 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 *Proc. ACM/IEEE Int. Conf. Computer-Aided Design*, 1997, pp. 91–98.

[10] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*: Kluwer, 1996.

[11] S. Liao and S. Devadas, "Solving covering problems using LPR-based lower bounds," in *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 117–120.

[12] V. M. Manquinho, P. F. Flores, J. P. Marques Silva, and A. L. Oliveira, "Prime implicant computation using satisfiability algorithms," in *Proc. IEEE Int. Conf. Tools with Artificial Intelligence*, Nov. 1997, pp. 232–239.

[13] D. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[14] G. L. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization*. New York: Wiley, 1988.

[15] J. P. Marques Silva and K. A. Sakallah, "GRASP: A new search algorithm for satisfiability," in *Proc. ACM/IEEE Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 220–227.

[16] T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Explicit and implicit algorithms for binate covering problems," *IEEE Trans. Computer-Aided Design*, vol. 16, no. 7, pp. 677–691, July 1997.

[17] S. Yang, "Logic synthesis and optimization benchmarks user guide," Microelectronics Center of North Carolina, Jan. 1991.

[18] H. Zhang, "SATO: An efficient propositional prover," in *Proc. Int. Conf. Automated Deduction*, July 1997, pp. 272–275.

Vasco M. Manquinho obtained the B.Sc. and M.Sc. degrees from the Technical University of Lisbon, Portugal, in 1996 and 1999, respectively. He is currently pursuing the Ph.D. degree at the Technical University of Lisbon.

Since 2001, he has been a Teaching Assistant at the Computer Science Department, Technical University of Lisbon, Portugal. His research interests include unate/binate covering, integer programming, and propositional satisfiability.

João P. Marques-Silva obtained the B.Sc. and M.Sc. degrees at the Technical University of Lisbon, Portugal, in 1988 and 1991, respectively, and the Ph.D. degree at the University of Michigan, Ann Arbor, in 1995.

Since 1995, he has been an Assistant Professor at the Computer Science Department, Technical University of Lisbon, Portugal, and a member of the Cadence European Laboratories. His research interests include algorithms for discrete optimization problems, namely satisfiability, unate/binate covering and integer programming, and applications of discrete optimization in EDA.