Lower Bounding Techniques for SAT-based Boolean Optimization

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

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

Abstract

This paper addresses the problem of integrating different lower bounding techniques into Satisfiability-based algorithms for the Binate Covering Problem (BCP), a well-known restriction of Boolean Optimization. The most significant aspect of integrating lower bounding techniques into Satisfiability-based algorithms for BCP is the ability to backtrack non-chronologically whenever lower bounding is applied. The lower bounding techniques considered include the well-known linear programming relaxations and maximum independent sets, among others. Besides establishing conditions for backtracking non-chronologically, we also develop conditions for simplifying the sets of explanations that are utilized for implementing non-chronological backtracking.

1 Introduction

The generic Boolean Optimization problem as well as several of its restrictions are well-known computationally hard problems, widely used as modeling tools in Computer Science and Engineering. These problems have been the subject of extensive research work in the past (see for example [1]). In this paper we address the Binate Covering Problem (BCP), one of the restrictions of Boolean Optimization. BCP can be formulated as the problem of finding a satisfying assignment for a given Conjunctive Normal Form (CNF) formula subject to minimizing a given cost function. As with generic Boolean Optimization, BCP also finds many applications, including the computation of minimum-size prime implicants, of interest in Automated Reasoning and Non-Monotonic Reasoning [12], and as a modeling tool in Electronic Design Automation (EDA) [3, 14].

The main objective of this paper is to describe how different lower bounding techniques can be incorporated into SAT-based branch-and-bound algorithms for the binate covering problem. Among other we study the utilization of liner programming relaxations and maximum independent sets, among others. For all the lower bounding techniques described we provide conditions that enable the search algorithm to backtrack non-chronologically whenever lower bounding is applied.

The paper is organized as follows. In Section 2 the notation used throughout the paper is introduced. Afterwards, we briefly review SAT-based branch-and-bound algorithms for BCP, and in section 4 describe different lower bounding procedures. In subsequent sections, we describe techniques for backtracking non-chronologically whenever lower bounding is applied. Moreover, we also provide conditions for simplifying the explanations that are used for backtracking non-chronologically.

Experimental results are presented in Section 6, and the paper concludes in Section 7.

2 Preliminaries

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

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

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

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

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

When a clause is unit (with only one unassigned literal) an assignment can be implied. For example, consider a propositional formula φ which contains clause $\omega = (x_1 + \bar{x}_2)$ and assume that $x_2 = 1$. For φ to be satisfied, x_1 must be assigned value 1 due to ω. Therefore, we say that $x_2 = 1$ *implies* $x_1 = 1$ due to ω or that clause ω *explains* the assignment $x_1 = 1$. These logical implications correspond to the application of the unit clause rule [5] and the process of repeatedly applying this rule is called *boolean constraint propagation* [13, 16]. It should be noted that throughout the remainder of this paper some familiarity with backtrack search SAT algorithms is assumed. The interested reader is referred to the bibliography (see for example [1, 13] for additional references).

Covering problems are often solved by branch-and-bound algorithms [4, 7, 14]. In these cases, each node of the search tree corresponds to a selected unassigned variable and the two branches out of the node represent the assignment of 1 and 0 to that variable. These variables are named *decision variables*. The first node is called the *root* (or the top node) of the search tree and corresponds to the *first decision level*. The decision level of each decision is defined as one plus the decision level of the previous decision.

3 Search Algorithms for Covering Problems

The most widely known approach for solving covering problems is the classical branch-and-bound procedure [14], 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 estimate 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, 7, 14] follow this approach.

The most commonly used lower bound estimation procedure for BCP is the approximation of a maximum independent set of clauses [3]. However, other procedures can be used, namely the ones based on linear-programming relaxations [7], Lagrangian relaxations [11] or the Log-approximation approach [3]. 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, different methods will be described with more emphasis on linear programming relaxations and the Log-approximation approach. Covering algorithms also incorporate several powerful reduction techniques, a comprehensive overview of which can be found in [3, 14].

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

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

The algorithm presented in [9] already incorporates the main pruning techniques of the GRASP SAT algorithm [13]. Hence, *bsolo* is a branch-and-bound algorithm for solving BCP that implements a non-chronological backtracking search strategy, clause recording and identification of necessary assignments. Mainly due to an effective conflict analysis procedure which allows non-chronological backtracking steps to be identified, *bsolo* performs better than other branch-and-bound algorithms in several classes of instances, as shown in [9]. However, non-chronological backtracking is limited to one specific type of conflict. In [10] is described how to apply non-chronological backtracking to *all* types of conflicts when using the approximation of a maximum independent set of clauses. In section 5 we describe how to apply the same concepts when using other lower bound estimation methods.

The main steps of a simplified version of the *bsolo* algorithm can be described as follows:

1. Initialize the upper bound to the highest possible value as defined (i.e. given by $ub = \sum_{j=1}^{n} c_j + 1$).

2. Start by checking whether the current state yields a conflict. This is done by applying boolean constraint propagation and, in case a conflict is reached, by invoking the conflict analysis procedure, recording relevant clauses and proceeding with the search procedure or backtrack if necessary.

3. If a solution to the constraints has been identified, update the upper bound according to $ub = \sum_{j=1}^{n} c_j \cdot x_j$. (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.)

4. Estimate a lower bound given the current variable assignments. If this value is higher than or equal to the current upper bound, issue a bound conflict and bound the search by applying the conflict analysis procedure to determine which decision node to backtrack to. Continue from step 2.

3.1 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 non-chronological manner).

However, the other type of conflict is handled differently. In *bsolo*, whenever a bound conflict is identified, a new clause *must* be added to the problem instance in order for a logical conflict to be issued and, consequently, to bound the search. This requirement is inherited from the GRASP SAT algorithm where, for guaranteeing completeness, both conflicts and implied variable assignments *must* be explained in terms of the existing variable assignments [13]. With respect to conflicts, each recorded conflict clause is built using the assignments that are deemed responsible for the conflict to occur. If the assignment $x_j = 1$ (or $x_j = 0$) is considered responsible, the literal $\bar{x}_j$ (respectively, literal x_j) is added to the conflict clause. This literal basically states that in order to avoid the conflict one possibility is certainly to have instead the assignment $x_j = 0$ (respectively, $x_j = 1$). Clearly, by construction, after the clause is built its state is unsatisfied. Consequently, the conflict analysis procedure has to be called to determine to which decision level the algorithm must backtrack to. Hence the search is bound.

Whenever a bound conflict is identified, one possible approach to building a clause to bound the search would be to include *all* decision variables in the search tree. In this case, the conflict would always depend on the last decision variable. Therefore, backtracking due to bound conflicts would necessarily be chronological (i.e. to the previous decision level), hence guaranteeing that the algorithm would be complete. Suppose that the set $\{x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1\}$ corresponds to all the search tree decision assignments and ω_{bc} is the clause to be added due to a bound conflict. Then we would have $\omega_{bc} = (\bar{x}_1 + x_2 + x_3 + \bar{x}_4)$. Again, the drawback of this approach (which was used in [9]) is that backtracking due to bound conflicts is always chronological, since it depends on all decision assignments made. In section 5 we propose a new procedure to build these clauses, which enables non-chronological backtracking due to bound conflicts.

4 Computation of Lower Bounds

The estimation of lower bounds on the value of the cost function is a very effective method to prune the search tree and the accuracy of lower bounding procedures is critical for identifying areas of the search space where solutions to the constraints with lower values of the cost function cannot be found. In this section we review methods commonly used to estimate a lower bound on the value of the cost function in instances of BCP.

4.1 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. The effectiveness of this method 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,

$$Cost(MIS) = \sum_{\omega \in MIS} Weight(\omega) \quad \text{where} \tag{2}$$

$$Weight(\omega) = \min_{x_j \in \omega} c_j \tag{3}$$

4.2 Linear Programming Relaxations

Linear programming relaxations have long been used as lower bound estimation procedures in branch-and-bound algorithms for solving integer programming problems [11]. For the binate covering problem, the utilization of linear programming relaxations as a lower bound estimation method is proposed in [7]. Moreover, it is also claimed that in most cases the linear programming relaxation (LPR) bound is higher than the one obtained with the MIS approach.

The general formulation of the LPR for a covering problem is obtained from (1) as follows:

$$\begin{array}{ll} \text{minimize} & z_{lpr} = \sum_{j=1}^{n} c_j \cdot x_j \\ \text{subject to} & A \cdot x \geq b \\ & x \geq 0 \end{array} \tag{4}$$

For simplicity the constraints $x \leq 1$ are not included. The solution of (1) is referred to as z^*_{cp}, whereas the solution of (4) is referred to as z^*_{lpr}.

It is well-known that the solution z^*_{lpr} of (4) is a lower bound on the solution z^*_{cp} of (1) [11]. Basically, any solution of (1) is also a feasible solution of (4), but the converse is not true. Moreover, for a given solution of (4) where $x \in \{0,1\}^n$, we necessarily have $z^*_{cp} = z^*_{lpr}$. Hence, the result follows. Furthermore, different linear programming algorithms can be used for solving (4), some of which with guaranteed worst-case polynomial run time [11].

```
greedy_solution(φ) {
  SOL = ∅;
  while (φ ≠ ∅) {
    var = min_{var∈Var(φ)} Cost(var) / Γ(cov_clauses(var, φ));
    φ = φ − cov_clauses(var, φ);
    SOL = SOL ∪ var;
  }
  return SOL;
}
```

Figure 1: Algorithm for computation of a greedy solution

4.3 Log-Approximation

It is well known that the MIS approach can be very far from the minimum cost solution in specific cases of problem instance matrixes [4]. Given that the approximation provided by the greedy algorithm is of poor quality, tighter lower bounds can be established. In [4] a new lower bound computation algorithm for unate covering is introduced which guarantees a logarithmic ratio bound on the minimum cost solution.

The algorithm in Fig. 1 describes a procedure for constructing a greedy solution for a covering problem with a set φ of constraints to satisfy. At each step a decision variable *var* is chosen, the clauses that become satisfied by *var* are removed from φ and a solution set is updated. Observe that the algorithm was conceived to tackle unate covering problems [4], but it can be easily modified in order to attempt to find greedy solutions for binate covering instances[1]. The variable to add to the solution set is the one which minimizes the relation between its cost and the value given by Γ. Let γ be a positive weighting function defined on a set of clauses (e.g. the number of free literals). We can define Γ as:

$$\Gamma(\varphi') = \sum_{\omega \in \varphi'} \gamma(\omega) \tag{5}$$

It can be shown [4] that based on the greedy solution given by the algorithm in Fig. 1, it is possible to obtain a lower bound on the covering problem which is log-approximable to the optimum value z^*_{cp}. This result is also valid for binate covering whenever the greedy algorithm is able to find a feasible solution. The lower bound is given by:

$$\frac{Cost(\texttt{SOL})}{r} \tag{6}$$

$$\text{where } r = \sum_{k=1}^{max_{\varphi'}\Gamma(\varphi')} 1/k \tag{7}$$

5 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 3.1, in the original *bsolo* algorithm non-chronological backtracking was only possible for logical conflicts. In the case of a bound conflict all the search tree decision assignments were used to explain the conflict.

[1] In binate covering, the greedy algorithm is unable to guarantee that a solution is found.

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. In [10] it is shown that these assignments are not in general associated with all decision levels in the search tree; hence non-chronological backtracking can take place. However, in [10], it is only described how to backtrack non-chronologically when a bound conflict occurs if the approximation of the maximum independent set of clauses (MIS) is used. In this paper we show that it is possible to have non-chronological backtracks when using linear-programming relaxations or the Log-approximation.

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

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

$$\omega_{cp} = \{l = \bar{x}_j : Cost(x_j) > 0 \wedge x_j = 1\} \tag{8}$$

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

We now consider $C.lower$. For simplicity suppose we are using the approximation of the maximum independent set of clauses as lower bound mechanism. Let MIS be the independent set of clauses, obtained by the method described in section 4.1, that determines the value of $C.lower$. 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 $\omega_i \in MIS$ these conditions only hold due to the literals in ω_i that are assigned value 0. If any of these literals was assigned value 1, ω_i would certainly not be in MIS since it would be a satisfied clause. Consequently, we can define a set of literals that explain the value of $C.lower$:

$$\omega_{cl} = \{l : l = 0 \wedge l \in \omega_i \wedge \omega_i \in MIS\} \tag{9}$$

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

$$\omega_{bc} = \omega_{cp} \cup \omega_{cl} \tag{10}$$

(Observe that the set union symbol in the previous equation denotes a disjunction of literals.) As long as this clause is unsatisfied, the values of $C.path$ and $C.lower$ will remain unchanged, and so the bound conflict will exist. We can thus use this unsatisfied clause ω_{bc} to analyze the bound conflict and decide where to backtrack to, using the conflict analysis procedure of GRASP [13]. We should observe that backtracking can be non-chronological, because clause ω_{bc} does not necessarily depend on all decision assignments. Moreover, due

to the clause recording mechanism, ω_{bc} can be used later in the search process to prune the search tree. If these clauses would depend on all decision assignments, clause recording would not be used since the same set of decision assignments is never repeated in the search process.

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

In order to build the bound conflict clause, we need to obtain the explanation for $C.lower$ (ω_{cl}). If different lower bound estimation procedures are used, different procedures are required for identifying an explanation for the bound conflict. In the remainder of this section we present a theoretical framework which also allows non-chronological backtracking when linear-programming relaxations or the Log-approximation are used.

5.1 Pruning with LP-Relaxation Lower Bounds

Linear programming relaxations (LPR) are a powerful method to estimate a lower bound value for instances of the binate covering problem [7]. However, the resulting backtrack from a bound conflict when using LPR has always been chronological. The naive approach to build a clause to bound the search in bound conflicts when using LPR would be to include all decision variables in the search tree. However, as stated in section 3.1, the resulting backtrack would necessarily be chronological. In this section, we present a new framework that allows non-chronological backtracking in bound conflicts when linear programming relaxations are used to estimate the lower bound value.

Remember that a bound conflict occurs when $C.path + C.lower \geq C.upper$, in which case a set of assignments that explains the conflict must be identified. Therefore, we must identify the assignments that explain the value on $C.path$ (ω_{cp}) and $C.lower$ (ω_{cl}) in order to build the bound conflict clause ω_{bc} to bound the search. Notice that the value on $C.path$ is independent on the lower bound computation procedure and the bound conflict clause can be built as in (8).

The approach to build ω_{cl} must be different from (9), since $C.lower$ depends on the value given by the LP-solver. Therefore, the information provided by the LP-solver must be used in order to backtrack non-chronologically in a lower bound conflict when using LPR for bound computation.

Given the value of $C.lower$ obtained with LPR as formulated in section 4.2, let S be the set of constraints with *slack* [2] variables assigned value 0. Observe that these are the constraints which actually limit the value of $C.lower$, and so will be referred to as the *active* constraints. When using LP-relaxations to compute the value of $C.lower$, the literals that assume value 0 in the active constraints are directly responsible for the value of $C.lower$. These literals correspond to the set ω_{cl} in the bound conflict clause. Applying this reasoning to both assignments of a given variable, allows implementing non-chronological backtracking and ω_{cl} can be build as:

$$\omega_{cl} = \{l : l = 0 \wedge l \in \omega_i \wedge \omega_i \in S\} \tag{11}$$

[2] See [8] for a definition of slack and artificial variables.

5.2 Reducing Dependencies in LPR Bound Conflicts

The previous section describes how to obtain an explanation on the value of *C.lower* when using LPR. However, a more careful analysis allows the identification of some situations where literals can be excluded from ω_{cl}, as presented in (11), since if they were to have a different value the conflict would still hold.

Suppose that in the solution of the LPR there is a variable $x_j = 1$ and let $S(x_j)$ denote the set of clauses from the active constraints set S where x_j appears. For every other literal $l_k \in S(x_j)$ we must have $l_k = 0$ in the LPR solution, since otherwise these were not active constraints. Remember that from (11) we say that all literals already assigned value 0 during the search process in clauses $S(x_j)$ are in ω_{cl}. However that might not always be the case.

Let $S^+(x_j)$ denote the clauses from $S(x_j)$ where, in the search process, all unassigned literals are positive. Therefore we must have $l_k = 0$ in the LPR solution for all other literals in $S^+(x_j)$. Notice that if we decrease the value of x_j, at least one x_i must be raised by the same value we decrease x_j in order to keep all constraints satisfied in the LPR solution. If we assume that all variables have the same cost in the cost function, decreasing the value of x_j would not decrease *C.lower* given by the LPR solution whenever $\|S^+(x_j)\| \geq 1$.

Notice that at least one clause in $S^+(x_j)$ is enough to justify the assignment $x_j = 1$ in the LPR solution. All other clauses in $S^+(x_j)$ may be considered irrelevant for the LPR solution in terms of the final cost, since the costly assignment can be justified by only one clause in $S^+(x_j)$. If we assume that $\omega_i \in S^+(x_j)$ is the one which justifies $x_j = 1$, all other clauses in $S^+(x_j)$ can be excluded from the computation of ω_{cl}. Choosing which clause ω_i from $S^+(x_j)$ justifies $x_j = 1$ can be implemented by a greedy procedure in order to have the smallest ω_{cl} as possible.

The same reasoning can easily be generalized for a set of assignments, instead of just the assignment of 1 to a single positive literal. Suppose that during the search process, we have an unresolved clause ω_i where all unassigned literals are positive. Given the LPR solution, if $\omega_i \in S$ then we must have a set V_m of m variables in ω_i such that $\sum_1^m x_j = 1$ and all other are assigned value 0, in order for this constraint to have a slack of 0. This is the necessary condition for $\omega_i \in S$.

One should note that ω_i is sufficient to justify the assignment to V_m. If we decrease the value of any of the m variables in V_m, another variable in the constraint must be raised by the same cost in order for ω_i to be satisfied. Either another variable in V_m or a variable assigned value 0 in the LPR solution. If we assume that all variables have the same cost in the cost function, decreasing the value of any of the variables in V_m would not result in a decrease on *C.lower* given by the LPR solution. Therefore, if there are other clauses in S different from ω_i which are also satisfied due to the assignments in V_m, then those other clauses in S are irrelevant for the final cost of the LPR solution and consequently irrelevant for the bound conflict.

Consider the following example where $\omega_1 = (\bar{x_1} + \bar{x_2})$, $\omega_2 = (x_1 + x_3)$, $\omega_3 = (x_2 + x_4)$, $\omega_4 = (x_2 + x_5)$ where $x_1 = x_4 = x_5 = 0, x_2 = x_3 = 1$ in the LPR solution. Notice that clause ω_3 *or* ω_4 (just one of them) can be considered irrelevant for the final cost of the LPR, as explained previously. If we analyze more carefully this situation we can also conclude that clause ω_1 (satisfied by $x_1 = 0$ in the LPR solution) can also be considered irrelevant since if x_1 was to have a different value, x_3 had to be assigned value 0 and the cost of the solution of the LPR would be the same.

Notice that if there is a clause $\omega_i \in S$ such that all unassigned literals are negative, for ω_i to be in S, there must be an assignment $x_j = 0$ while all other variables are assigned value 1 in the LPR solution. One should note that the assignments of 1 to the other constraint variables is not because of ω_i, but due to other problem constraints. However, ω_i might be constraining the set of the LPR solution cost by making that at least one of its variables must be assigned value 0. Nevertheless, that is not always the case and in certain circumstances, ω_i does not have to be considered in the ω_{cl} calculation.

Suppose we have a clause $\omega_i \in S$ such that all unassigned literals are negative (like ω_1 in our example) and there is another clause $\omega_k \in S$ where all unassigned literals in ω_k are positive (like ω_2). If we have in the LPR solution $x_j = 0$ satisfying ω_i and x_j appears at most in one clause ω_k (clause with only unassigned positive literals), then ω_i can be considered irrelevant to the bound explanation. If x_j would have a different value (raising the cost of the LPR solution), it could only be balanced with lowering the literal which satisfies ω_k. Since we are supposing that all literals have the same cost, the assignment $x_j = 0$ does not constrains the overall cost of the LPR solution. Considering again our example, if there was another clause $\omega_5 = (x_1 + x_6)$ where $x_6 = 1$ in the LPR solution, if x_1 could be assigned value 1, both $x_3 = x_6 = 0$ would be possible assignments and the LPR solution would be lower. Therefore, in this case, ω_1 was essential in order to justify the LPR solution and consequently the bound conflict. That is why x_1 can only appear at most in one clause with all unassigned positive literals in the search process.

5.3 Pruning with Log-Approximation Lower Bounds

When *C.lower* is estimated using the Log-approximation method described in section 4.3, its value depends on the variable assignments chosen by the greedy algorithm (see Fig. 1). Notice that each time a variable assignment is chosen, it depends on the value of function Γ. Therefore, choosing a variable assignment depends on the clauses which become satisfied with that assignment.

Suppose an assignment to variable x_j is chosen at iteration k of the greedy algorithm. This assignment is due to the fact that there is a set of clauses given by $cov\_clauses(x_j, \varphi(k)')$ [3] which become satisfied and this set of clauses allows variable x_j to be chosen by the algorithm. Therefore, the clauses in *cov_clauses* provide an explanation for the assignment to variable x_j. Moreover, the literals assigned value 0 in $cov\_clauses(x_j, \varphi(k)')$ are the ones deemed responsible since if they were to have a different value, $cov\_clauses(x_j, \varphi(k)')$ would be a smaller set which could cause the assignment to variable x_j not to be required and hence *C.lower* could be lower. Notice that if any of the literals considered responsible were to satisfy some of these clauses by having the opposite value, the set of clauses to satisfy (given by *cov_clauses*) would be smaller and a lower value for *C.lower* could be obtained, possibly solving the bound conflict situation.

Let *C.lower* be estimated using the Log-approximation method and let SOL be the solution found by the greedy algorithm in n iterations which yields a bound conflict. In that case, a bound conflict clause ω_{bc} must be created to bound the search. The explanation on *C.path* is determined as described previously in section 5, since it does not depend on the lower bound estimation method. Moreover, the explanation on *C.lower* is given by:

$$\omega_{cl} = \{l : l = 0 \wedge l \in \omega_i \wedge \omega_i \in \pi(n)\} \tag{12}$$

[3] Notice that $\varphi(k)'$ denotes the set of clauses still to satisfy at iteration k of the greedy algorithm.

		lp-solve	cplex	scherzo	opbdp	bsolo (MIS)	bsolo (LPR)
Benchmark	min.	CPU	CPU	CPU	CPU	CPU	CPU
aim-100-1_6-yes1-2	100	–	–	–	1104.5	0.28	0.24
aim-100-2_0-yes1-3	100	–	–	235.84	12.14	0.26	0.27
aim-100-3_4-yes1-4	100	–	–	11.56	0.19	0.68	11.46
aim-200-1_6-yes1-3	200	–	–	–	–	0.41	1.56
aim-200-3_4-yes1-1	200	–	–	–	9.60	2.86	172.95
aim-50-1_6-yes1-1	50	757.3	113.4	0.76	0.02	0.06	0.09
aim-50-2_0-yes1-2	50	1284.5	107.6	1.81	0.09	0.11	1.10
ii8a1	54	162.8	63.0	0.33	0.62	0.52	2.77
ii8a2	–	ub 149	ub 147	–	ub 141	ub 140	ub 140
ii8b1	191	ub 243	840.4	–	ub 191	1042.19	517.2
ii8c1	–	ub 364	ub 304	–	ub 302	ub 302	ub 302
jnh12	94	–	2251.7	0.87	0.01	0.25	0.31
jnh17	95	–	842.5	4.90	0.06	0.88	61.40
jnh7	89	ub 89	ub 90	1.49	0.10	0.63	120.23
ssa7552-158	1327	ub 1327	ub 1328	14.54	ub 1327	3.26	842.25
ssa7552-160	1359	ub 1359	ub 1359	–	ub 1359	9.26	ub 1359

Table 1: Algorithm comparison

where $\pi(n)$ is the set of all clauses covered by the assigned variables chosen until iteration n which is equivalent to:

$$\pi(n) = cov\_clauses(SOL[1], \varphi(1)') \cup ... \cup cov\_clauses(SOL[n], \varphi(n)') \tag{13}$$

where $SOL[k]$ is the selected assignment at iteration k in the greedy algorithm.

Notice that at iteration n all clauses from φ (clauses that are not yet satisfied) are in $\pi(n)$, since all are covered at iteration n of the greedy algorithm. Nevertheless, it is possible that the resulting bound conflict clause ω_{bc} does not depend on the last decision assignment level and non-chronological backtracking can take place.

6 Experimental Results

In this section we include experimental results of several algorithms in two different sets of benchmarks. The first table present results for instances of the MCNC benchmark set [15], whereas the remaining tables present results for instances of 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 [9, 12].

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. 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 was either due to the time or memory limits imposed. Besides the time taken and the number of decisions made to solve the instances, it is also shown the number of non-chronological backtracks and the highest jump made in the search tree.

The experimental procedure consisted of running a selected set of problem instances with *bsolo* and several other algorithms. In table 1 we can observe the results of several algorithms on instances of the minimum-size prime implicant problem. Clearly, *lp_solve* [2]

		bsolo (a)				bsolo (b)			
Benchmark	min.	CPU	Dec.	NCB	Jump	CPU	Dec.	NCB	Jump
aim-100-1_6-yes1-2	100	0.28	82	17	11	0.24	82	17	11
aim-100-2_0-yes1-3	100	2.46	97	21	8	0.27	74	15	3
aim-100-3_4-yes1-4	100	11.16	88	8	4	11.46	88	8	4
aim-200-1_6-yes1-3	200	1.58	56	9	8	1.56	56	9	8
aim-200-3_4-yes1-1	200	163.73	201	21	5	172.95	201	21	5
aim-50-1_6-yes1-1	50	0.07	34	7	7	0.09	34	7	7
aim-50-2_0-yes1-2	50	0.78	48	8	6	1.10	54	10	3
aim-50-3_4-yes1-3	50	1.92	34	4	3	1.93	34	4	3
ii8a1	54	2.67	72	1	2	2.77	72	1	2
ii8b1	191	493.41	245	1	2	517.2	245	0	1
jnh12	94	0.33	14	2	2	0.31	14	2	2
jnh17	95	63.42	65	7	2	61.40	65	7	2
jnh7	89	119.23	34	2	2	120.23	34	2	2
ssa7552-158	1327	810.45	382	0	1	842.25	382	0	1
ssa7552-159	1327	2192.52	605	0	1	2052.18	605	0	1

Table 2: Non-chronological backtracking using LPR (1)

		bsolo (a)				bsolo (b)			
Benchmark	min.	CPU	Dec.	NCB	Jump	CPU	Dec.	NCB	Jump
5xp1.b	12	708.21	967	1	2	656.52	942	79	7
9sym.b	5	133.14	94	1	2	126.05	94	4	3
alu4.b	–	ub51	9007	0	1	ub51	9007	10	4
apex4.a	–	ub788	1981	0	1	ub788	2186	8	3
bench1.pi	121	450.03	453	1	2	438.72	453	7	2
clip.b	15	48.68	413	1	2	45.97	413	1	2
count.b	24	7.85	16	1	2	8.52	16	0	1
e64.b	–	ub48	105395	0	1	ub48	84403	343	9
f51m.b	18	127.92	849	1	2	111.45	849	50	4
jac3	15	372.18	102	1	2	374.55	102	6	25
rot.b	–	ub121	30765	0	1	ub121	40738	5	5
sao2.b	25	324.67	1824	1	2	238.35	1824	17	2

Table 3: Non-chronological backtracking using LPR (2)

and *cplex* (generic Integer Linear Programming solvers) are unable to solve almost all instances given the time limit. Notice that only for a few problem instances was it able to find an upper bound. *scherzo* [4], a state-of-the-art BCP solver that incorporates several powerful pruning techniques in a classical branch-and-bound algorithm, is also unable to solve most of the example instances. The SAT-based linear search algorithm *opbdp* [1] is able to solve most instances, hence suggesting that these instances are well-suited for SAT-based solvers. Notice however that the two versions of *bsolo* presented here (using the approximation of the maximum independent set (MIS) or the linear-programming relaxation (LPR) as lower bounding mechanism) are able to solve almost every instance or give a better upper bound on the optimum solution. These two versions differ significantly in terms of time performance since in most cases the LP-solver incorporated in *bsolo (LPR)* was very slow in solving the LPR. Nevertheless, because it provides a better lower bound estimation, *bsolo (LPR)* makes fewer decisions than *bsolo (MIS)*.

Table 2 shows how *bsolo* behaves using LPR as a lower bound procedure in a small set of solvable instances. In *bsolo(a)* all bound-based conflicts backtrack chronologically, while in *bsolo(b)* we apply the method presented in section 5 to explain the bound conflict. In these instances, most of the non-chronological backtracks are from logical conflicts and not from

		bsolo(LPR)		bsolo(MIS)		scherzo	
Benchmark	min.	CPU	Dec.	CPU	Dec.	CPU	Dec.
5xp1.b	12	656.52	942	181.02	1640	4.5	2234
9sym.b	5	126.05	94	27.91	135	3.6	320
alu4.b	–	ub51	time	ub 51	time	–	time
apex4.a	776	ub788	time	ub 781	time	87.4	48359
bench1.pi	121	438.72	453	ub 123	time	–	time
clip.b	15	45.97	413	67.09	1313	0.6	97
count.b	24	8.52	16	12.27	102	478.0	299780
e64.b	–	ub48	time	ub 48	time	–	mem.
f51m.b	18	111.45	849	97.00	1671	1.9	1586
jac3	15	374.55	102	ub 17	time	4.9	292
rot.b	–	ub121	time	ub 120	time	–	time
sao2.b	25	238.35	1824	9.58	281	0.9	279

Table 4: Results for *bsolo* and *scherzo*

bound-based ones. It was rarely observed non-chronological backtracks due to bound-based conflicts. For a different benchmark set, as shown in table 3, we can observe that when using *bsolo* (b) non-chronologically jumps due to bound conflicts occur in the search tree, improving the algorithms performance.

In instances of the MCNC benchmark set, *scherzo* is considered to be one of the best and fastest solvers. In table 4 we can observe the results of both *scherzo* and *bsolo* (using LPR and MIS). Notice that *bsolo (LPR)* can be slower than *bsolo (MIS)* for some instances but *bsolo (LPR)* makes fewer decisions than *bsolo (MIS)* since the LPR provides a much tighter lower bound than the approximation of the maximum independent set (MIS). By integrating linear-programming relaxations as a lower bound procedure, *bsolo (LPR)* is able to solve a larger set of instances with less search effort. However, due to the fact that the LP-solver is not yet properly integrated in *bsolo*, the overall time can be higher in some instances.

7 Conclusions

This paper extends known search pruning techniques, from the Boolean Satisfiability domain, to branch-and-bound algorithms for solving the Binate Covering Problem. We present conditions that allow for non-chronological backtracking in the presence of bound conflicts when different lower bounding procedures are utilized. Among others, the lower bounding procedures considered include linear programming relaxations and maximum independent sets.

Previous work was already done regarding the integration of linear programming relaxations in boolean optimization algorithms. However, this is the first time an algorithm using this lower bounding mechanism is augmented with the ability for backtracking non-chronologically in the presence of bound conflicts. Moreover, we have established conditions for reducing the size of bound conflict explanations, which further elicits non-chronological backtracking. Preliminary results obtained on several instances of the Binate Covering Problem indicate that the proposed techniques can reduce the amount of search, that can potentially result in a more competitive algorithm.

Future research work will naturally include seeking further simplification of the bound clauses created, applying techniques already used for other lower bounding procedures, namely the approximation of maximum independent set of clauses. We can also preview

the generalization of the conditions we present for different variable costs. In addition, and for obtaining more competitive experimental results, a more adequate integration of the LP package with the search algorithm needs developed.

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

[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*, pages 197–202, June 1996.

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

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

[7] S. Liao and S. Devadas. Solving Covering Problems Using LPR-Based Lower Bounds. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 117–120, 1997.

[8] J. J. J. M. S. Bazaraa and H. D. Sherali. *Linear Programming and Network Flows.* 2nd Ed., John Wiley & Sons, 1989.

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

[10] V. M. Manquinho and J. P. Marques-Silva. Conditions for non-chronological backtracking in boolean optimization. In *AAAI Workshop on the Integration of AI and OR Techniques for Combinatorial Optimization,*, August 2000.

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

[12] C. Pizzuti. Computing Prime Implicants by Integer Programming. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, pages 332–336, November 1996.

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

[14] 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.

[15] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide. Microelectronics Center of North Carolina, January 1991.

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