Journal of Automated Reasoning (2005)
DOI: 10.1007/s10817-005-9005-y

Heuristic-Based Backtracking Relaxation for Propositional Satisfiability

ATEET BHALLA*, INÊS LYNCE, JOSÉ T. DE SOUSA
and JOÃO MARQUES-SILVA
Technical University of Lisbon, IST/INESC-ID, Rua Alves Redol 9, 1000-029, Lisbon, Portugal.
e-mail: {ateet.bhalla, ines.lynce, jts, jpms}@inesc-id.pt

Abstract. In recent years backtrack search algorithms for propositional satisfiability (SAT) have been the subject of dramatic improvements. These improvements allowed SAT solvers to successfully solve instances with thousands or tens of thousands of variables. However, many new challenging problem instances are still too hard for current SAT solvers. As a result, further improvements to SAT technology are expected to have key consequences in solving hard real-world instances. This paper introduces a new idea: choosing the backtrack variable using a heuristic approach with the goal of diversifying the regions of the space that are explored during the search. The proposed heuristics are inspired by the heuristics proposed in recent years for the decision branching step of SAT solvers, namely, VSIDS and its improvements. Completeness conditions are established, which guarantee completeness for the new algorithm, as well as for any other incomplete backtracking algorithm. Experimental results on hundreds of instances derived from real-world problems show that the new technique is able to speed SAT solvers, while aborting fewer instances. These results clearly motivate the integration of heuristic backtracking in SAT solvers.

1. Introduction

Propositional satisfiability is a well-known NP-complete problem, with theoretical and practical significance and with extensive applications in many fields of computer science and engineering, including artificial intelligence and electronic design automation.

Current state-of-the-art SAT solvers incorporate sophisticated pruning techniques as well as new strategies for organizing the search. Effective search pruning techniques are based, among others, on no-good learning and dependency-directed backtracking [24] and back-jumping [8], whereas recent effective strategies introduce variations on the organization of backtrack search. Examples of such strategies are weak-commitment search [25], search restarts [12], and random backtracking [15, 20].

Advanced techniques applied to backtrack search SAT algorithms have achieved remarkable improvements [2, 11, 18, 19], having been shown to be

* Author for correspondence.

crucial for solving hard instances of SAT obtained from real-world applications. Moreover, and from a practical perspective, the most effective algorithms are *complete* and so are able to prove unsatisfiabiltiy. Indeed, this is often the objective in a large number of significant real-world applications.

Nevertheless, it is also widely accepted that local search has some advantages compared to backtrack search. Although it is debatable which are the real advantages of local search (e.g., see [7]), one of them seems to be the use of search restarts. Search restarts prevent the search form *getting stuck* in a locally optimal partial solution. The advantage of search restarts has motivated the study of approaches for relaxing backtracking conditions (while still ensuring completeness). The key idea is to *unrestrictedly* choose the point to backtrack to, in order to avoid thrashing, that is, exploring useless portions of search space corresponding to very similar conflicting sets of assignments. Moreover, one can think of combining different forms of relaxing the identification of the backtrack point.

In this paper, we propose to use heuristic knowledge to select the backtrack point. Besides describing the generic heuristic backtracking search strategy, we establish backtracking heuristics inspired by the most effective branching heuristics proposed in recent years, namely, the VSIDS heuristic used by Chaff [19] and the BerkMin's branching heuristic [11].

Simply replacing deterministic backtracking with heuristic backtracking in SAT algorithms has two major drawbacks: (1) the resulting algorithm is no longer complete, and (2) an algorithm applying heuristic backtracking for every backtrack step becomes very unstable.

To eliminate these drawbacks, we introduce the concept of unrestricted backtracking algorithms. Each backtrack step is either a complete form of backtracking (i.e., chronological or nonchronological backtracking) or an incomplete form of backtracking (e.g., heuristic backtracking). Clearly an unrestricted backtracking algorithm applying heuristic backtracking after every k steps (with $k > 1$) and nonchronological backtracking every other steps is more stable than an unrestricted backtracking algorithm applying heuristic backtracking for every backtrack step. Moreover, we establish completeness conditions for unrestricted backtracking algorithms. These conditions guarantee completeness for *any* instantiation of the unrestricted backtracking algorithm.

This paper extends previous work. We first introduced our heuristic backtracking ideas in [3], where we showed that heuristic backtracking is superior to other forms of unrestricted backtracking such as search restarts and random backtracking. In [4], we introduced the completeness conditions and modified the algorithm accordingly to make it complete. Some preliminary and promising results have been presented in [3] and in [4]. This paper gives a more comprehensive description of the different forms of backtracking and integrates heuristic backtracking within the framework of unrestricted backtracking. In addition, we present improved experimental results that show that the benefits of heuristic backtracking increase for hard-to-solve problem instances.

We summarize the contributions of this paper as follows: (1) we introduce heuristic backtracking algorithms; (2) we show that heuristic backtracking is a special case of unrestricted backtracking, and we describe different approaches for guaranteeing completeness of unrestricted backtracking; and (3) we give experimental results that indicate that the proposed heuristic backtracking algorithm is a competitive approach for solving real-world instances of SAT.

The remainder of this paper is organized as follows. The next section presents definitions used throughout the paper. In Section 3 we briefly survey backtrack search SAT algorithms. In Section 4 we introduce heuristic backtracking. Section 5 describes unrestricted backtracking algorithms for SAT and explains how heuristic backtracking can be regarded as a special case or unrestricted backtracking. In addition, we address completeness issues. Section 6 gives experimental results, and Section 7 describes related work. In Section 8, we conclude the paper and give directions for future research work.

2. Definitions

This section introduces the notational framework used throughout the paper. Propositional variables are denoted $x_1, \ldots, x_n$ and can be assigned truth values 0 (or F) or 1 (or T). The truth value assigned to a variable x is denoted by $v(x)$. (When clear from context we use $x = v_x$, where $v_x \in \{0,1\}$). A literal l is either a variable x_i or its negation $\neg x_i$. A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses. A clause is said to be *satisfied* if at least one of its literals assume value 1, *unsatisfied* if all of its literals assume value 0, *unit* if all but one literal assume value 0 and *unresolved* otherwise. Literals with no assigned truth value are said to be *free literals*. A formula is said to be *satisfied* if all its clauses are satisfied, and is *unsatisfied* if at least one clause is unsatisfied. A *truth assignment* for a formula is a set of pairs of variables and their corresponding truth values. The SAT problem consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied.

SAT algorithms can be characterized as being either *complete* or *incomplete*. Complete algorithms can establish unsatisfiablity if given enough CPU time; incomplete algorithms cannot. Examples of complete and incomplete algorithms are backtrack search and local search algorithms, respectively. In a search context, complete algorithms are often referred to as *systematic*, whereas incomplete algorithms are referred to as *nonsystematic*.

3. Backtrack Search SAT Algorithms

Over the years a large number of algorithms have been proposed for SAT, from the original Davis–Putnam procedure [6], to recent backtrack search algorithms [2, 11, 18, 19] and local search algorithms [23], among many others.

The vast majority of backtrack search SAT algorithms are built on the original backtrack search algorithm of Davis, Logemann, and Loveland [5]. The backtrack search algorithm is implemented by a *search process* that implicitly enumerates the space of 2^n possible binary assignments to the n variables of the problem. Each different truth assignment defines a *search path* within the search space. A *decision level* is associated with each variable selection and assignment. (The notation $x@d$ is used to denote that variable x has been assigned at decision level d.) The first variable selection corresponds to decision level 1, and the decision level is incremented by 1 for each new decision assignments.[★] In addition, and for each decision level, the *unit clause rule* [6] is applied. The iterated application of the unit clause rule is often referred to as Boolean constraint propagation (BCP). If a clause is unit, then the sole free literal must be assigned value 1 for satisfying the formula. In this case, the values of the literal and of the associated variable are said to be *implied*. Thus, assigned variables can be distinguished as *decision variables* and *implied variables*.

In chronological backtracking, the search algorithm keeps track of which decision assignments have been toggled. Given an unsatisfied clause (i.e., a *conflict* or a *dead end*) at decision level d, the algorithm checks whether at the current decision level the corresponding decision variable x has already been toggled. If not, the algorithm erases the variable assignments that are implied by the assignment on x, including the assignment on x, assigns the opposite value to x, and marks decision variable x as toggled. In contrast, if the value of x has already been toggled, the search backtracks to decision level $d - 1$.

Recent state-of-the-art SAT solvers utilize different forms of nonchronological backtracking [2, 18, 19]. In these algorithms each identified conflict is analyzed to identify the variable assignments that caused it, and a new clause (*no-good*) is created to explain and prevent the identified conflicting conditions from happening again. The created clause is then used to compute the backtrack point as the *most recent* decision assignment represented in the recorded clause; moreover, some of the (larger) recorded clauses are eventually deleted. Clauses can be deleted opportunistically whenever they are no longer *relevant* for the current search path [18].

Figure 1 illustrates the differences between chronological backtracking (CB) and the nonchronological backtracking (NCB). On the top of the figure appears a generic search tree (either possible in the context of CB or NCB). The search is performed according to a depth-first search, and therefore the non-dashed branches define the search space explored so far. On the one hand, and when a conflict is found, the chronological backtracking algorithm makes the search backtrack to the most recent, yet untoggled decision variable (see CB(a)). On the

[★] All assignments made before the first decision assignment correspond to decision level 0, a preprocessing step.

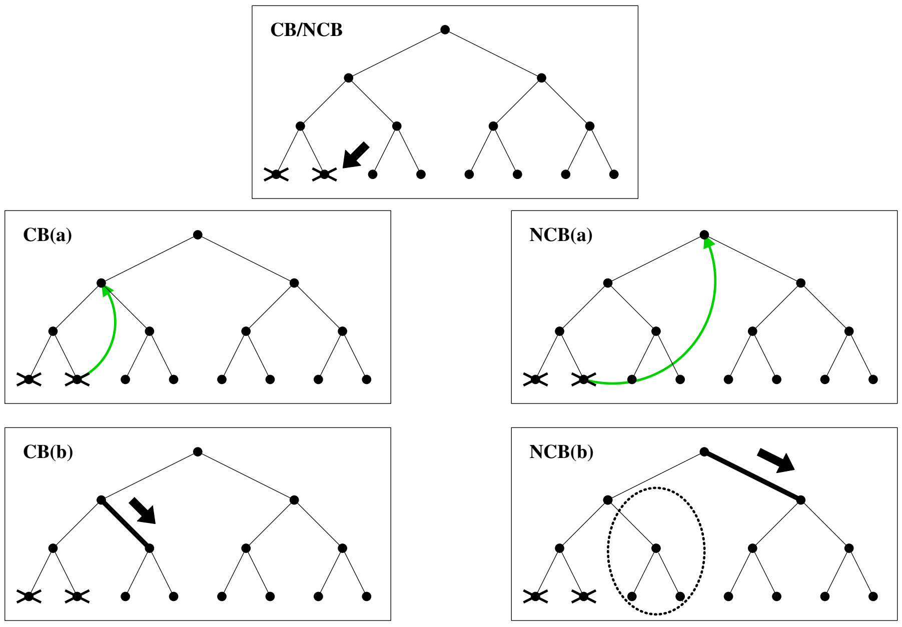

Figure 1. Chronological backtracking (CB) vs nonchronological backtracking (NCB).

other hand, when nonchronological backtracking is applied, the backtrack point is computed as the *most recent* decision assignment from all the decision assignments represented in the recorded clause. In this case the search backtracks to a higher level in the search tree (NCB(a)), skipping portions of the search tree that are found to have no solution (see NCB(b)). From the final figures (CB(b) and NCB(b)) it is plain to conclude that the number of nodes explored by NCB is always equal or smaller than the number of nodes explored by CB.[★] (Observe that no-goods can also reduce the search space because *similar* conflict paths of the search space are avoided in the future).

4. Heuristic Backtracking

Heuristic backtracking consists of selecting the backtrack point in the search tree using a heuristic function of the variables in the most recently recorded clause. Different heuristic functions can be envisioned for applying heuristic backtracking. In this work we implemented three heuristics:

1. *Plain heuristic*: uses a simple heuristic function.

[★] Assuming that a fixed-order branching heuristic is used.

2. *VSIDS-like heuristic*: inspired by the VSIDS branching heuristic used by Chaff [19].
3. *BerkMin-like heuristic*: inspired by the BerkMin's branching heuristic [11].

In all cases the backtrack point is computed as the variable with the largest heuristic metric. Next, we describe how the three approaches are implemented in a heuristic backtracking algorithm.

4.1. PLAIN HEURISTIC BACKTRACKING

After a conflict (i.e., an unsatisfied clause) is identified, a *conflict clause* is created. The conflict clause is then used for *heuristically* deciding which decision assignment is to be toggled. Observe that when a conflict clause is created, all the literals in the clause are assigned value 0. This fact motivates the search to backtrack to the most recent decision level with implications on the conflict clause.

Under the plain heuristic backtracking approach, the search is allowed to backtrack to *any* decision level with implications on the literals of the conflict clause. The backtrack point (i.e., decision level) is computed by selecting the decision level with the largest number of occurrences (assigned or implied literals) in the newly recorded clause. In addition, ties are broken randomly. This approach contrasts with the usual nonchronological backtracking approach, in which the most recent decision variable with implications on the conflict is selected as backtrack point.

EXAMPLE 1. Suppose that plain heuristic backtracking is to be applied after recording clause $\omega = (x_1 \vee x_3 \vee \neg x_5 \vee \neg x_9 \vee x_{12})$. Also, suppose that each literal in ω has been assigned at a given decision level: $\omega = (x_1@10 \vee x_3@7 \vee \neg x_5@8 \vee \neg x_9@7 \vee x_{12}@2)$. Clearly, the decision level with the largest number of occurrences (in this case 2 occurrences) is decision level 7. Hence, plain heuristic backtracking makes the search backtrack to level 7.

4.2. VSIDS-LIKE HEURISTIC BACKTRACKING

The second approach to heuristic backtracking is based in the variable-state independent decaying sum (VSIDS) branching heuristic. The heuristic [19]. VSIDS was the first of a new generation of decision heuristics. This heuristic has been used in *Chaff*, a highly optimized SAT solver. More than to develop a well-behaved heuristic, the motivation in Chaff has been to design a fast heuristic. In fact, one of the key properties of this strategies is the low computational overhead, due to being independent of the variable state. As a result, the variable metrics are updated only when there is a conflict.

Similarly to Chaff, in our VSIDS-like backtracking heuristic we have a counter for each literal. Each counter is initialized with the number of occurrences of the literal in the formula. Moreover, each counter is incremented when a new conflict clause containing the literal is added to the clause database. In addition, after every 255 decisions, the metric values are divided by a constant factor of 2, to give preference to variables occurring in the latest conflict clauses. With our VSIDS-like backtracking heuristic, whenever a conflict occurs, the literal in the just recorded clause with the highest metric is used to select the backtrack point.

EXAMPLE 2. Suppose that the VSIDS-like heuristic backtracking is to be applied after recording clause $\omega = (x_1@10 \vee x_3@7 \vee \neg x_5@8 \vee \neg x_9@7 \vee x_{12}@2)$. In addition, suppose that the VSIDS metric for a given variable x is given by $vsids(x)$ and that $vsids(x_1) = 45$, $vsids(x_3) = 5$, $vsids(x_5) = 94$, $vsids(x_9) = 32$ and $vsids(x_{12}) = 41$. The literal in the just recorded clause with the highest metric is x_5. Hence, the VSIDS-like backtracking heuristic makes the search backtrack to level 8, that is, the level where x_5 was assigned.

4.3. BERKMIN-LIKE HEURISTIC BACKTRACKING

The third approach for implementing heuristic backtracking is inspired by the BerkMin's branching heuristic [11], which, in turn, has been inspired by the VSIDS heuristic used in Chaff. In the BerkMin's branching heuristic, the process for updating the metrics of the literals is different. On the one hand, in Chaff the current activity of a variable x is computed by counting the number of occurrences of x in the conflict clause. On the other hand, in BerkMin a wider set of clauses involved in causing the conflict is taken into account for computing each variable's activity. This procedure avoids overlooking some variables that do not appear in the conflict clause, while actively contributing to the conflict.

In our BerkMin-like backtracking heuristic, we increment the metrics of the literals in all clauses that are directly involved in producing the conflict. The metrics are updated during the process of conflict analysis, which can find all clauses involved in producing the conflict by traversing an implication graph data structure. This process finishes with the creation of the conflict clause. As in the case of the VSIDS-like backtracking heuristic, the literal in the conflict clause with the highest metric is used to select the backtrack point.

EXAMPLE 3. Consider again the clause given in Example 2: $\omega = (x_1@10 \vee x_3@7 \vee \neg x_5@8 \vee \neg x_9@7 \vee x_{12}@2)$. Also, suppose that the values given for the BerkMin's metric are given by function *berkmin* and that $berkmin(x_1) = 31$, $berkmin(x_3) = 38$, $berkmin(x_5) = 2$, $berkmin(x_9) = 15$ and $berkmin(x_{12}) = 53$. The literal in the just recorded clause with the highest metric is x_{12}. Hence, BerkMin-like heuristic backtracking makes the search backtrack to level 2, that is, the level where x_{12} has been assigned.

5. Unrestricted Backtracking

Heuristic backtracking can be viewed as a special case of unrestricted backtracking [16]. While in unrestricted backtracking any form of backtrack step can be applied, in heuristic backtracking the backtrack point is computed from heuristic information, obtained from the current and past conflicts.

Unrestricted backtracking algorithms allow the search to *unrestrictedly* backtrack to *any* point in the current search path whenever a conflict is reached. Besides the freedom for selecting the backtrack point in the decision tree, unrestricted backtracking allows the application of different types of backtrack steps. Each backtrack step can be selected among chronological backtracking, nonchronological backtracking, (e.g., search restarts, weak-commitment search, random backtracking, or heuristic backtracking). More formally, unrestricted backtracking (UB) allows the application of a sequence of backtrack steps $\{\mathrm{BSt}_1, \mathrm{BSt}_2, \mathrm{BSt}_3, \ldots\}$ such that each backtrack step BSt_i can be a chronological (CB), a nonchronological (NCB), or an incomplete form of backtracking (IFB). This formalism allows capturing the backtracking search strategies used by state-of-the-art SAT solvers [2, 11, 18, 19]. Indeed, if the backtracking sequence consists of always applying chronological backtracking steps or always applying nonchronological backtracking steps, then we capture the chronological and nonchronological backtracking search strategies, respectively.

Unrestricted backtracking gives a unified representation for different backtracking strategies, which allows establishing general completeness conditions for *classes* of backtracking strategies. This is more convenient than analyzing each individual strategy, as has been done in [22, 25]. In what follows, we establish general completeness conditions for unrestricted backtracking, which are valid for any special case of unrestricted backtracking; this includes heuristic backtracking, the main thrust of this paper.

Figure 2 exemplifies how an incomplete form of backtracking can lead to incompleteness, by providing possible sequels to the search process shown in Figure 1. Three backtracking strategies are illustrated: chronological (CB), nonchronological (NCB) and incomplete form of backtracking (IFB). The search path that leads to the solution is marked with the letter **S**. For CB and NCB the solution is found by orderly exploring the search space. With IFB the search backtracks to *any* point, which may cause skipping the search subspace that leads to the solution. Hence, something must be done to ensure the correctness and completeness of an unrestricted backtracking algorithm that includes incomplete backtracking steps. First, and similar to local search, we have to assume that variable toggling in unrestricted backtracking is *reversible*.

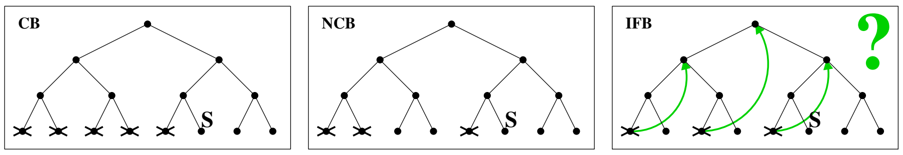

Figure 2. Comparing chronological backtracking (CB), nonchronological backtracking (NCB) and incomplete forms of backtracking (IFB).

This means that the solution can be found later, even if the solution is skipped during the search. Irreversible variable toggling would yield an incorrect or incomplete algorithm. Second, with reversible variable toggling, we must ensure that the algorithm terminates or otherwise it may loop forever in the search space.

A number of techniques can be used to ensure the completeness of unrestricted backtracking algorithms. These techniques are analyzed in [16] and reviewed in the remainder of this section. Completeness techniques for unrestricted backtracking can be organized in two classes:

– Marking recorded clauses as nondeletable. This solution may yield an exponential growth in the number of recorded clauses.[★]
– Increasing a given constraint (e.g., the number of nondeletable recorded clauses) in between applications of different backtracking schemes. This solution can be used to guarantee a polynomial growth of the number recorded clauses.

5.1. COMPLETENESS ISSUES

It has been explained above how unrestricted backtracking can yield incomplete algorithms. Hence, it is important to be able to apply conditions that guarantee the completeness for each newly devised SAT algorithm that utilizes IFB Steps.

The results presented in this section generalize completeness results that have been proposed in the past (for specific backtracking relaxations) to UB. We start by establishing a few already known results, and then we establish additional results for UB.

In what follows we assume the organization of a backtrack search SAT algorithm as described earlier in this paper. The main loop of the algorithm consists of selecting a decision variable, assigning the variable, and propagating the assignment by using BCP. If an unsatisfied clause occurs (i.e., a *conflict*) the

[★] In practice an exponential growth in the number of recorded clauses hardly ever arises.

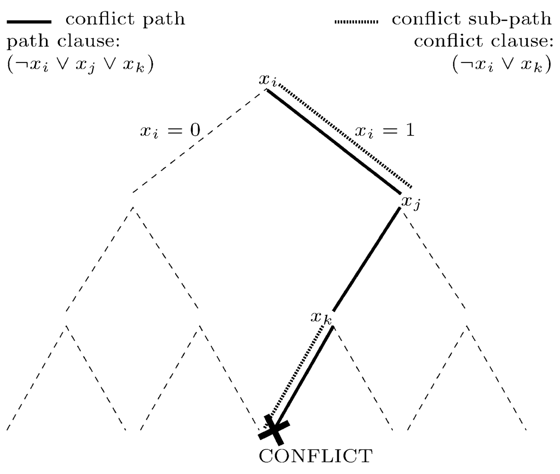

Figure 3. Search tree definitions.

algorithm backtracks to a decision assignment that can be toggled.[★] Each time a conflict is identified, all the current decision assignments define a *conflict path* in the search tree. (We restrict the definition of conflict path solely with respect to the decision assignments.) After a conflict is identified, we may apply a *conflict analysis* procedure [2, 18, 19] to identify a subset of the decision assignments that represent a sufficient condition for producing the same conflict. The subset of decision assignments that is declared to be associated with a given conflict is referred to as a *conflict subpath*. A straightforward conflict analysis procedure consists of construction a clause with *all* the decision assignments in the conflict path. In this case the created clause is referred to as a *path-clause*. Figure 3 illustrates these definitions. We can now established a few general results that will be used throughout this section.

PROPOSITION 1. *If an unrestricted backtracking search algorithm does not repeat conflict paths, then it is complete.*

Proof. Assume a problem instance with n variables. Observe that there are 2^n possible conflict paths. If the algorithm does not repeat conflict paths, then it must necessarily terminate.

PROPOSITION 2. *If an unrestricted backtracking search algorithm does not repeat conflict subpaths, then it does not repeat conflict paths.*

Proof. If a conflict subpath is not repeated, then no conflict path can contain the same subpath, and so no conflict path can be repeated.

[★] Without loss of generality, we assume that NCB uses irreversible variable toggling after backtracking. In some recent algorithms this happens as an implication caused by the newly derived conflict clause [19].

PROPOSITION 3. *If an unrestricted backtracking search algorithm does not repeat conflict subpaths, then it is complete.*

Proof. Given the two previous results, if no conflict subpaths are repeated, then no conflict paths are repeated, and so completeness is obtained.

PROPOSITION 4. *If the number of times an unrestricted backtracking search algorithm repeats conflict paths or conflict subpaths is upperbounded by a constant, then the backtrack search algorithm is complete.*

Proof. We prove the result for conflict paths; the proof for conflict subpaths is similar. Let M be a constant denoting an upper bound on the number of times a given conflict path can be repeated. Since the total number of distinct conflict paths is 2^n, and since each can be repeated at most M times, then the total number of conflict paths the backtrack search algorithm can enumerate is $M \times 2^n$, and so the algorithm is complete.

PROPOSITION 5. *For an unrestricted backtracking search algorithm following holds:*

1. If the algorithm creates a path clause for each identified conflict, then the search algorithm repeats no conflict paths.
2. If the algorithm creates a conflict clause for each identified conflict, then the search algorithm repeats no conflict subpaths.
3. If the algorithm creates a conflict clause (or a path clause) after every M identified conflicts, then the number of times an unrestricted backtracking search algorithm repeats conflict sub-paths (or conflict paths) is upper-bounded.

In all of the above cases, the search algorithm is complete.

Proof. Recall that the search algorithm always applies BCP after making a decision assignment. Hence, if a clause describing a conflict has been recorded and not deleted, BCP may trigger the same conflict with a different set of decision assignments. As a result, conflict paths are not repeated. The same holds true for conflict sub-paths. Since neither conflict paths nor conflict subpaths are repeated, the search algorithm is complete (form Propositions 1 and 3). With respect to creating (and recording) a conflict clause (or a path clause) after every M identified conflicts, clearly the number of times a given conflict subpath (or conflict path) is repeated is upper-bounded. Hence, using the results of Proposition 4 completeness is guaranteed.

Observed that Proposition 5 holds *independently* of which backtrack step is take each time a conflict is identified. Hence, as long as we record a conflict for each identified conflict, *any* form of unrestricted backtracking yields a complete algorithm. Less general formulations of this result have been proposed in the recent past [9, 22, 25].

The results established so far guarantee completeness at the cost of recording (and keeping) a clause for each identified conflict. Next, we propose and analyze conditions for relaxing this requirement. As a result, we allow for some clauses to be deleted during the search process and require only some specific recorded clauses to be kept.[★] (We note that clause deletion does not apply to chronological backtracking strategies and that existing clause deletion policies for nonchronological backtracking strategies do not compromise the completeness of the algorithm [18].) We also propose other conditions that do not require specific recorded clauses to be kept.

PROPOSITION 6. *An unrestricted backtracking algorithm is complete if it records (and keeps) a conflict-clause for each identified conflict for which an IFB step is taken.*

Proof. At most 2^n IFB steps can be taken because a conflict clause is recorded for each identified conflict after an IFB step is taken. Hence, conflict subpaths due to IFB steps cannot be repeated. Moreover, additional backtrack steps that may be applied (CB and NCB) also ensure completeness. Hence, the resulting algorithm is complete.

PROPOSITION 7. *Given an integer constant M, an unrestricted backtracking algorithm is complete if it records (and keeps) a conflict-clause after every M identified conflicts for which an IFB step is taken.*

Proof. The result immediately follows from the Propositions 5 and 6.

Under the conditions above, the number of recorded clauses grows linearly with the number of conflicts after IFB steps. Thus the number of recorded clauses is worst-case exponentially in the number of variables.

Other approaches to guarantee completeness involve increasing the value of some constraint associated with the search algorithm. The following results illustrate these approaches.

PROPOSITION 8. *Suppose that an unrestricted backtracking strategy applies a sequence of backtrack steps. If for this sequence the number of conflicts between IFB steps is allowed to increase strictly after each IFB step, then the resulting algorithm is complete.*

Proof. If only CB or NCB steps are taken, then the resulting algorithm is complete. When the number of conflicts in between IFB steps reaches 2^n, the algorithm is guaranteed to terminate.

We note that this result can be viewed as a generalization of the completeness condition used in search restarts, which consists of increasing the backtrack

[★] We say that a recorded clause is *kept* provided it is prevented from being deleted during the subsequent search.

cutoff value after search restart [1].[★] Also observe that in this situation the growth in the number of clauses can be made polynomial, provided clause deletion is applied on clauses recorded form NCB and IFB steps.

The next result establishes conditions for guaranteeing completeness in algorithms that opportunistically delete recorded clauses (as a result of an IFB step). The idea is to increase the size of the recorded clauses that are kept after each IFB step. Another approach is to increase the life-span of large recorded clauses, by increasing the relevance-based learning threshold [2].

PROPOSITION 9. *Suppose that an unrestricted backtracking strategy applies a specific sequence of backtrack steps. If, for this sequence, either the size of the largest recorded clause kept or the size of the relevance-based learning threshold is strictly increased after each IFB step is taken, then the resulting algorithm is complete.*

Proof. When either the size of the largest recorded clause reaches value n or the relevance-based learning threshold reaches value n, all recorded clauses will be kept, and so completeness is guaranteed from Proposition 5.

Observe that for this last result the number of clauses can grow exponentially with the number of variables. Moreover, we note that the observation regarding the increase of the relevance-based learning threshold was first suggested in [19].

One final result addresses the number of times conflict paths and conflict subpaths can be repeated.

PROPOSITION 10. Under the conditions of Proposition 8 and Proposition 9, the number of times a conflict path or a conflict subpath is repeated is upper-bounded.

Proof. The resulting algorithms are complete and thus known to terminate after a maximum number of backtrack steps (which is constant for each instance). Hence, the number of times a conflict path (or conflict subpath) can be repeated is necessarily upper-bounded.

5.2. HEURISTIC BACKTRACKING UNDER THE UNRESTRICTED BACKTRACKING FRAMEWORK

Unrestricted backtracking provides a framework for combining different forms of backtracking. These forms of backtracking may be complete, incomplete, or a combination of both. The completeness conditions established for unrestricted backtracking hold regardless of the comprised forms of backtracking.

We have noted before that applying heuristic backtracking at every backtrack step may lead to very unstable algorithms. Conversely, keeping all the recorded clauses to avoid this instability may lead to a significant memory overhead.

[★] Given this condition, the resulting algorithm resembles iterative-deepening.

Hence, the solution adopted for this problem is to combine heuristic backtracking with other complete forms of backtracking.

In what follows we refer to heuristic backtracking as an instantiation of unrestricted backtracking where incomplete heuristic backtracking steps are combined with complete nonchronological backtracking steps. For each algorithm, we will specify the frequency of the heuristic backtracking steps and the heuristic used. As mentioned before, we have developed three backtracking heuristics: the plain, VSIDS-like, and BerkMin-like backtracking heuristics.

6. Experimental Results

This section presents experimental results of applying heuristic backtracking to different classes of problem instances. We compare heuristic backtracking with nonchronological backtracking and nonchronological backtracking combined with search restarts [12], one of the most effective backtracking relaxation schemes known to date. Search restarts are now part of the most competitive backtrack search SAT algorithms [19, 11], and our goal here has been to demonstrate that heuristic backtracking is a more competitive form of backtracking relaxation.

The algorithms have been experimentally evaluated by using the JQuest2 SAT solver [17]. JQuest2 is a competitive solver and has been ranked among the top solvers in the industrial category in the SAT 2003 competition.* JQuest2 has been implemented in Java for providing an integrated framework for rapid prototyping of SAT algorithms.

It offers a significantly faster development time for testing new ideas in SAT algorithms, but its overall performance is slower than a C or C++ implementation because of the overhead associated with the Java virtual machine. It has been demonstrated that JQuest2 is slower than Chaff by an average factor of 2 [17]. The CPU time limit for each instance was set to 10^4 s. All experiments were run on the same P4/1.7 Ghz/1 GByte of RAM Linux machine.

Different SAT algorithm prototypes have been implemented and compared. The algorithms differ only in the unrestricted backtracking strategy applied. Five backtracking strategies are compared:

1. Plain heuristic backtracking.
2. VSIDS-like heuristic backtracking.
3. BerkMin-like heuristic backtracking.
4. Search restarts.
5. Nonchronological backtracking.

All algorithms use the VSIDS decision branching heuristic. In choosing a decision or backtrack variable, a slight randomization is used to select among the

* http://www.satlive.org/SATCompetition/2003/.

variables with the best metrics provided by the different heuristics. Combining the values of the metrics with a certain degree of randomization is known to produce good results.

The algorithms have been applied to 14 classes of problem instances containing 320 problem instances in total. In NCB, a nonchronological backtrack step is performed every step. In the other algorithms is defined as follows: an incomplete form of backtracking step (HB or restarts) is performed after every $10^4 + i \times 10^3$ backtracks, where i is incremented every time an IFB step is performed. The increase of constant i and the fact that conflict derived clauses are marked undeletable guarantee the completeness of the algorithms.

Table I shows the results obtained for each class of instances. *#I* denotes the number of problem instances, *Dec* denotes the average number of decision nodes per instance, *Time* denotes the average CPU time per instance, and *X* denotes the number of aborted instances. In addition, each column indicates a different form of backtracking relaxation:

– HB(P) indicates the plain heuristic backtracking algorithm is applied after $10^4 + i \times 10^3$ backtracks, where i is incremented every time a HB step is taken.
– HB(V) indicates the VSIDS-like heuristic backtracking algorithm is applied after $10^4 + i \times 10^3$ backtracks, where i is incremented every time a HB step is taken.
– HB(B) indicates the BerkMin-like heuristic backtracking algorithm is applied after $10^4 + i \times 10^3$ backtracks, where i is incremented every time a HB step is taken.
– RST indicates that search restarts are applied after $10^4 + i \times 10^3$ backtrack, where i is incremented every time a search restart is taken.
– NCB indicates nonchronological backtracking is applied in every back-track step.

From the results in Table I several observations and comments can be made.

HB algorithms abort fewer instances. An instance is aborted whenever the memory or CPU time constraint is reached. In these experiments all instance abortions have been caused by memory exhaustion, which shows that fewer clauses using HB as compared to search restarts. A possible explanation is that our heuristics are more likely to reuse information provided by earlier conflicts than is the search restarts algorithm, which is more prone to encounter new conflict clauses after each restart. Equivalently, one can say that HB favors a more local search rather than search restarts.

The nonchronological backtracking algorithm is not a competitive approach, in terms of both decisions and CPU time. This is true when compared with any of the other four algorithms. In addition, the search restarts algorithm seems to be the second worst approach, although more competitive than the nonchronological backtracking algorithm. The computed average speedup against the nonchronological backtracking algorithm for the set of instances used is $1.95\times$.

Table I. Performance of different algorithms.

Benchmarks	#*I*	HB(P)			HB(V)			HB(B)			RST			NCB		
		Dec	Time	*X*	Dec	Time	*X*	Dec	Time	*X*	Dec	Time	*X*	Dec	Time	*X*
bmc-barrel	8	1307492	4070.22	0	1013787	1819.11	0	584680	735.61	0	1031297	2397.81	0	1339798	4789.36	0
bmc-queueinvar	10	84784	33.02	0	85666	42.04	0	69758	25.86	0	66713	20.93	0	117823	69.34	0
bmc-longmult	16	1112467	5868.87	3	937649	3088.83	3	946813	4276.77	3	1177463	7873.19	5	1490667	8156.69	5
sss-sat-1.0	100	3470750	3018.83	0	3005868	1785.38	0	1486274	877.58	0	3142078	2425.27	0	3527717	4029.94	0
sss-1.0	48	939535	658.87	0	681025	199.68	0	467650	93.97	0	736851	408.56	0	945588	1167.82	0
sss-1.0a	9	191838	230.4	0	176227	108.77	0	98217	37.5	0	198363	459.68	0	260389	858.87	0
fvp-unsat.1.0	4	191300	180.93	1	196671	217.28	1	102657	38.94	1	167714	109.31	1	222725	395.18	1
qg	22	402257	1762.75	0	381790	1344.69	0	236258	577.63	0	283459	950.08	0	494616	2829.94	0
Beijing	16	522884	5055.56	2	509764	4063.42	2	517462	4906.94	2	523849	5284.38	2	585194	5653.04	2
equiv-checking	25	2317494	2035.26	2	2355508	2282.49	2	913835	1101.26	2	2853280	3467.24	2	3307203	4163.99	2
par16	10	74641	41.12	0	72607	27.8	0	56619	18.1	0	111228	79.95	0	122614	106.59	0
des-encryption	32	533520	3801.09	2	512640	3194.06	2	480812	2885.39	2	578128	5005.26	2	784206	9055.62	2
satplan_sat	11	78777	79.45	0	47403	33.1	0	28682	27.05	0	58412	71.42	0	127017	102.39	0
satplan_unsat	9	39042	65.5	0	27371	41.2	0	10021	24.3	0	51502	89.54	0	56780	95.68	0

The plain heuristic backtracking algorithm performed slightly better on average than the search restarts algorithm. Although these results are not very conclusive, they seem to indicate that using some heuristic information when performing backtracking is better than not using any information at all, as is the case of search restarts. Moreover, in the next table it is shown that, when applied to some instances, the plain backtracking heuristic is significantly superior to search restarts and nonchronological backtracking.

The VSIDS-like heuristic backtracking algorithm performed better than the search restarts algorithm for most of the instances, in terms of both the number of decisions and CPU time, even though slower in performance on some test instances. Its computed average speedup against the search restarts algorithm for the set of instances used is 1.77×. (Note that this is a lower bound of the average speedup, since the instances aborted by the search restarts algorithm are a superset of the instances aborted by the VSIDS-like heuristic backtracking algorithm; the aborted instances have not been taken into account in computing the average speedup).

The BerkMin-like heuristic backtracking algorithm performed better than the VSIDS-like heuristic backtracking algorithm. This result is consistent with the fact that the BerkMin decision branching heuristic is generally superior to the VSIDS decision backtracking heuristic. Its computed average speedup against the search restarts algorithm for the set of instances used is 3.32×.

Given the large number of instances tested, these results clearly demonstrate the backtracking heuristic can speed up execution time for the classes of problems tested. It is also remarkable that their effect is similar to the effect of decision branching heuristic: if a heuristic A is better than a heuristic B for decision branching, then A is also better than B for backtracking.

The result presented in Table I are significantly better than the preliminary results previously presented in [4]. The reason is that we eliminated many easy-to-solve instances from each problem class. These instances do not benefit from heuristic backtracking or search restarts because they can be solved quickly before a significant number of IFB steps are applied, if any. Large instances do benefit from HB or restarts because these techniques help get out of dead-ends in the search tree. Hence, they should be applied infrequently. In our studies we concluded that, similar to search restarts, HB is best when applied once in every 10^4 backtracks. More frequent applications cause the algorithms to wander without focusing in regions of search space that need a more thorough exploration. When applied infrequently, HB allows finding a solution or proving unsatisfiability using a significantly lower number of decisions.

To show that the performance of the heuristics improve with the hardness of the problem instances, we manually selected a set of 18 harder-to-solve instances.

The results in Table II show that for the set of harder-to-solve instances the benefits of heuristic backtracking are more visible. The three HB algorithms performs better than the search restarts algorithm and nonchronological backtracking, which aborted two of the instances (marked with *).

Clearly, the search restarts algorithm performs better than the nonchronological backtracking algorithm, in terms of both the number of decisions and CPU time.

The plain heuristic backtracking algorithm performed better than both the search restarts algorithm and the nonchronological backtracking algorithm for most of the instances. This is true both in terms of the number of decisions and CPU time.

The VSIDS-like heuristic backtracking algorithm performed better than the search restarts algorithm, both in terms of the number of decisions and CPU time. Its average speed-up has been computed as greater than $2.62\times$.

The BerkMin-like heuristic backtracking algorithm was again the best of the three backtracking algorithms. Its average speed-up against the search restarts algorithm has been computed as greater than $9.63\times$.

As can be concluded from the experimental results, heuristic backtracking can yield significant savings in CPU time, allows significant reductions in the number of decision nodes and also allows for a smaller number of instances to be aborted. This is true for several of the classes of problem instances analyzed.

7. Related Work

Dependency-directed backtracking and no-good learning were originally proposed by Stallman and Sussman in [24] in the area of truth maintenance systems. In the area of constraints satisfaction problems (CSPs), the topic was independently studied by J. Gaschnig [8] and others (see, e.g., [21]) as different forms of backjumping.

The introduction of relaxations in the backtrack steps is also related to dynamic backtracking [9]. Dynamic backtracking establishes a method by which backtrack points can be moved deeper in the search tree. This avoids the unneeded erasing of the amount of search that has been done thus far. The objective is to find a way to directly "erase" the value assigned to a variable as opposed to backtracking to it, moving the backjump variable to the end of the partial solution in order to replace its value without modifying the values of the variables that currently follow it. More recently, Ginsberg and McAllester combined local search and dynamic in an algorithm that enables arbitrary search movement [10], starting with *any complete assignment* and evolving by flipping values of variables obtained from the conflicts.

Local search and dynamic backtracking have also been combined by Prestwich in the Constrained Local Solver (CLS) [20]. CLS is constructed by randomizing the backtracking component of a systematic algorithm: that is, allowing backtracking to occur on *arbitrarily chosen* variables. The new algorithm has the drawback of being incomplete.

Table II. Performance of different algorithms on individual instances

Benchmarks	Instance	HB(P)		HB(V)		HB(B)		RST		NCB	
		Dec	Time	Dec	Time	Dec	Time	Dec	Time	Dec	Time
bmc	barrel9	869896	3332.68	780650	2542.17	238566	391.1	707033	1903.58	790029	266051
bmc	longmult10	229373	1720.57	220432	1234.4	188654	573.41	*	*	*	*
bmc	longmult15	284658	2500.61	175422	672.68	219087	1487.77	*	*	*	*
sss-sat-1.0	2dlx_...bug056	56197	34.88	55240	31.86	24565	19.48	60345	48.23	63303	56.32
sss.1.0a	dlx2_...bug54	29744	36.71	24466	33.21	2304	11.60	36596	58.94	35018	53.01
sss.1.0	dlx2_cl	17144	9.65	12882	4.92	13206	6.95	33244	14.13	36781	15.98
fvp-unsat.1.0	2dlx_ca_...bp_f	36156	30.36	32294	19.24	31181	15.42	42027	40.47	47982	83.67
qg	qg2-08	137102	762.72	36265	47.48	41455	278.52	58271	554.23	67954	608.59
qg	qg5-13	97086	298.67	103891	771.32	77634	186.81	51839	116.46	80370	217.36
equiv-checking	c7552	198240	95.92	243101	145.48	151021	75.93	313592	293.61	318831	461.34
equiv-checking	c7552-s	295689	158.52	400875	347.84	175070	58.91	411626	594.24	444888	867.96
equiv-checking	c3540_bug	2211	1.34	2560	2.69	557	0.38	3378	4.95	8472	15.24
des-encryption	cnf-r3-b1-k1.1	9943	6.25	10937	9.3	4716	1.11	13068	18.75	13960	21.65
des-encryption	cnf-r3-b1-k2.2	2275	1.18	5566	2.82	1220	0.76	7792	3.36	8042	4.76
par16	par16-1-c	8866	3.65	8019	2.70	7273	1.3	11143	18.8	21773	25.74
par16	par16-4	1964	2.4	1117	1.65	887	0.4	3378	3.55	3386	4.48
satplan_sat	bw-large.d	34890	62.13	22438	27.84	18805	17.09	30855	53.37	49110	82.55
satplan_unsat	logistics.c	5458	3.9	5018	2.61	4211	1.02	5334	3.4	5754	6.49

In weak-commitment search [25], the algorithm constructs a consistent partial solution but commits to the partial solution *weakly*. In weak-commitment search, whenever a conflict is reached, the *whole* partial solution is abandoned, in explicit contrast to standard backtracking algorithm where the most recently added variable is removed form the partial solution.

Moreover, search restarts have been proposed and shown effective for hard instances of SAT [12]. The search is repeatedly restarted whenever a cutoff value is reached. In [1], search restarts were jointly used with learning for solving hard real-world instances of SAT. This latter algorithm is complete because the backtrack cutoff value increases after each restart. One additional example of backtracking relaxation is described in [22], which is based on attempting to construct a complete solution, that restarts each time a conflict is identified. More recently, highly optimized complete SAT solvers [11, 19] have successfully combined nonchronological backtracking and search restarts, again obtaining remarkable improvements in solving real-world instances of SAT.

Other algorithms are known for performing an overall local search while using systematic search to prune the search space. For example, Jussien and Lhomme introduced the path-repair algorithm for CSP [14], which adds domain filtering techniques and no-good learning to local search. Furthermore, Hirsch and Kojevnikov introduced the UnitWalk SAT solver [13], which combines the iterative application of the unit clause rule with local search.

8. Conclusions and Future Work

This paper proposes the utilization of heuristic backtracking in backtrack search SAT solvers. The proposed algorithm, based on heuristic knowledge, is presented in the context of a backtracking-based SAT algorithm, which is currently the most successful class of general-purpose SAT algorithms especially for real-world applications. The most well-known branching heuristic used in state-of-the-art SAT solvers were adapted to the backtrack step of SAT solvers. The experimental results illustrate the usefulness of heuristic backtracking and realize the potential of this technique on practical examples, especially those coming from real-world applications.

The main contributions of this paper can be summarized as follows:

1. A new heuristic backtracking search SAT algorithm is proposed that heuristically selects the point to backtrack to.
2. The proposed algorithm is shown to be a special case of unrestricted backtracking, and different approaches for ensuring completeness are described.
3. Experimental results indicate that significant savings in search effort can be obtained for different organizations of the proposed heuristic backtrack search algorithm.

In fact, hundreds of problems instances have been analyzed in this paper, where heuristic backtracking algorithms have been compared to a state-of-the-art SAT solver algorithm. The only difference between the new algorithms and the reference SAT solver is the backtracking step: the new algorithms apply heuristic backtracking steps instead of search restarts, the best form of incomplete backtracking known to date.

Three backtracking heuristics have been tested: a plain heuristic that uses information from the conflict-clause, a VSIDS-like heuristic, and a BerkMin-like heuristic. Our results show that the better the heuristic is for decision branching, the more useful it is for backtracking, which is a consistent result.

In a set of 320 instances, the best backtracking heuristic (BerkMin's) shows an average speedup of about $3.5\times$ as compared with the search restarts algorithm. For a set of 18 harder-to-solve instances, the heuristic backtracking algorithms have been able to solve all of them, while the search restarts algorithm and nonchronological backtracking aborted two instances.

The heuristic backtracking procedure developed in this work is now ready to be incorporated in SAT solvers, with guaranteed performance improvement.

For future work, a more comprehensive experimental evaluation is required for combining different forms of decision heuristics and backtracking relaxation algorithms, thus motivating the utilization of multiple search strategies in backtrack search SAT algorithms.

Acknowledgements

This work is partially supported by the European research project IST-2001-34607 and by Fundação para a Ciência e Tecnologia under research projects POSI/CHS/34504/2000, POSI/SRI/41926/2001 and POSI/EIA/61852/2004.

References

1. Baptista, L. and Marques-Silva, J. P.: Using randomization and learning to solve real-world instances of satisfiablility, in R. Dechter (ed.), *Proceedings of the International Conference of Principles and Practice of Constraint Programming*, Vol. 1894 of Lecture Notes in Computer Science, 2000, pp. 489–494.
2. Bayardo Jr., R. and Scharg, R.: Using CSP look-back techniques to solve real-world SAT instances, in *Proceedings of the National Conference on Artificial Intelligence*, 1997, pp. 203–208.
3. Bhalla, A., Lynce, I., de Sousa, J. and Marques-Silva, J.: Heuristic backtracking algorithms for SAT, in *Proceedings of the International Workshop of Microprocessor Test and Verification*, 2003, pp. 69–74.
4. Bhalla, A., Lynce, J., de Sousa, J. and Marques-Silva, J. P.: Heuristic-based backtracking for propositional satisfiability, in F. Moura-Pires and S. Abreu (eds.), *Proceedings of the Portuguese Conference on Artificial Intelligence*, Vol., 1894 of Lecture Notes in Artificial Intelligence, 2003, pp. 116–130.

5. Davis, M., Logemann, G. and Loveland, D.: A machine program for theorem proving, *Commun. Assoc. Comput. Mach.* **5** (1962), 394–397.
6. Davis, M. and Putnam, H.: A computing procedure for quantification theory, *J. Assoc. Comput. Mach.* **7** (1960), 201–215.
7. Freuder, E. C., Dechter, R., Ginsberg, M. L., Selman, B. and Tsang, E.: Systematic versus stochastic constraint satisfaction, in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995, pp. 2027–2032.
8. Gaschnig, J.: Performance Measurement and Analysis of Certain Search Algorithms, PhD thesis, Carnegie-Mellon University, Pittsburgh, PA.
9. Ginsberg, M. L.: Dynamic backtracking, *J. Artif. Intell. Res.* **1** (1993), 25–46.
10. Ginsberg, M. L. and McAllester, D.: GSAT and dynamic backtracking, in *Proceedings of the International Conference of Principles of Knowledge and Reasoning*, 1994, pp. 226–237.
11. Goldberg, E. and Nonikov, Y.: BerkMin: A Fast and Robust SAT-Solver, in *Proceedings of the Design and Test in Europe Conference*, 2002, pp. 142–149.
12. Games, C. P., Selman, B. and Kautz, H.: Boosting combination search through randomization, in *Proceedings of the National Conference on Artificial Intelligence*, 1998, pp. 431–437.
13. Hirsch, E. A. and Kojevnikov, A.: Solving Boolean satisfiability using local search guided by unit clause elimination, in *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 2001, pp. 605–609.
14. Jussien, N. and Lhomme, O.: Local search with constraint propagation and conflict-based heuristics, in *Proceedings of the National Conference on Artificial Intelligence*, 2000, pp. 169–174.
15. Lynce, I., Baptista, L. and Marques-Silva, J. P.: Stochastic systematic search algorithm for satisfiability, in *Proceedings of the LICS Workshop on Theory and Applications of Satisfiability Testing*, 2001, pp. 1–7.
16. Lynce, I. and Marques-Silva, J. P.: Complete unrestricted backtracking algorithms for satisfiability, in *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, 2002, pp. 214–221.
17. Lynce, I. and Marques-Silva, J. P.: On implementing more efficient SAT data structures, in *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, 2003, pp. 510–516.
18. Marques-Silva, J. P. and Sakallah, K. A., GRASP—A search algorithm for propositional satisfiability, *IEEE Trans. Comput.* **48**(5) (1999), 506–521.
19. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. and Malik, S.: Engineering an efficient SAT solver, in *Design Automation Conference*, 2001, pp. 530–535.
20. Prestwich, S.: A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences, in R. Dechter (ed.), *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Vol. 1894 of Lecture Notes in Computer Science, 2000, pp. 337–352.
21. Prosser, P.: Hybrid algorithms for the constraint satisfaction problems, *Comput. Intell.* **9**(3) (1993), 268–299.
22. Richards, E. T. and Richards, B.: Non-systematic search and no-good learning, *J. Autom. Reason.* **24**(4) (2000), 483–533.
23. Selman, B. and Kautz, H.: Domain-independent extensions to GSAT: Solving large structured satisfiability problems, in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1993, pp. 290–295.
24. Stallman, R. M. and Sussman, G. J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artif. Intell.* **9** (1977), 135–196.
25. Yokoo, M.: Weak-commitment search for solving satisfaction problems, in *Proceedings of the National Conference on Artificial Intelligence*, 1994, pp. 313–318.