Annals of Mathematics and Artificial Intelligence **37:** 307–326, 2003.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

An overview of backtrack search satisfiability algorithms

Inês Lynce and João P. Marques-Silva

Technical University of Lisbon, IST/INESC/CEL, Rua Alves Redol, 9, 1000-029 Lisboa, Portugal
E-mail: {ines,jpms}@sat.inesc.pt

Propositional Satisfiability (SAT) is often used as the underlying model for a significant number of applications in Artificial Intelligence as well as in other fields of Computer Science and Engineering. Algorithmic solutions for SAT include, among others, local search, backtrack search and algebraic manipulation. In recent years, several different organizations of local search and backtrack search algorithms for SAT have been proposed, in many cases allowing larger problem instances to be solved in different application domains. While local search algorithms have been shown to be particularly useful for random instances of SAT, recent backtrack search algorithms have been used for solving large instances of SAT from real-world applications. In this paper we provide an overview of backtrack search SAT algorithms. We describe and illustrate a number of techniques that have been empirically shown to be highly effective in pruning the amount of search on significant and representative classes of problem instances. In particular, we review strategies for non-chronological backtracking, procedures for clause recording and for the identification of necessary variable assignments, and mechanisms for the structural simplification of instances of SAT.

Keywords: satisfiability, search algorithms, backtracking

1. Introduction

Propositional Satisfiability is a well-known NP-complete problem, with extensive applications in Artificial Intelligence (AI), Electronic Design Automation (EDA), and many other fields of Computer Science and Engineering. In recent years several competing solution strategies for SAT have been proposed and thoroughly investigated [23]. Local search algorithms [40,41] have allowed solving extremely large satisfiable instances of SAT. These algorithms have also been shown to be very effective in randomly generated instances of SAT. On the other hand, several improvements to the backtrack search Davis–Putnam algorithm have been introduced. These improvements have been shown to be crucial for solving large instances of SAT derived from real-world applications and for proving unsatisfiability [3,35]. It is interesting to note that proving unsatisfiability is the final objective in several applications including, among others, automated theorem proving in AI, and circuit verification and circuit delay computation in EDA [38]. One final approach consists of algebraic simplification of propositional formulas. Even though not competitive in practice, algebraic simplification techniques have been used as a preprocessing step in several recently proposed SAT algorithms [14,22,27].

Despite the potential interest of all these algorithmic solutions, we believe that further improvements to backtrack search algorithms for SAT can have significant practical impact in many areas of Computer Science and Engineering, in particular those where local search cannot in general be applied, e.g., in proving unsatisfiability. In this paper we propose to overview backtrack search algorithms for SAT, giving a particular emphasis to the techniques that are commonly used for pruning the search. Moreover, we also analyze other techniques, highly effective in other application domains, and which, given their simplicity, can easily be incorporated in backtrack search SAT algorithms.

The paper is organized as follows. We start in the next section by introducing the notational framework used throughout the paper. In section 3, we describe SAT algorithms based on backtrack search. Next, in section 4, we explain decision making heuristics. Techniques commonly used for pruning the amount of search are reviewed in section 5, whereas other less well-known techniques are described in section 6. Experimental evidence of the practical application of backtrack search SAT algorithms is given in section 7. Afterwards, section 8 analyzes recent work in SAT domain.[1] Finally, section 9 concludes the paper by providing some perspective on future work on backtrack search SAT algorithms.

2. Definitions

A conjunctive normal form (CNF) formula φ on n binary variables $x_1, \ldots, x_n$ is the conjunction of m clauses $\omega_1, \ldots, \omega_m$ each of which is the disjunction of one or more literals, where a literal is the occurrence of a variable x or its complement x'. A formula denotes a unique n-variable Boolean function $f(x_1, \ldots, x_n)$ and each of its clauses corresponds to an implicate of f. Clearly, a function f can be represented by many equivalent CNF formulas. The satisfiability problem (SAT) is concerned with finding an assignment to the arguments of $f(x_1, \ldots, x_n)$ that makes the function equal to 1 or proving that the function is equal to the constant 0.

A backtrack search algorithm for SAT is implemented by a *search process* that implicitly enumerates the space of 2^n possible binary assignments to the problem variables. During the search, a variable whose binary value has already been determined is considered to be *assigned*; otherwise it is *unassigned* with an implicit value of $X \equiv \{0, 1\}$. A *truth assignment* for a formula ω is a set of assigned variables and their corresponding binary values. It will be convenient to represent such assignments as sets of pairs of variables and their assigned values; for example, $A = \{(x_1, 0), (x_7, 1), (x_{13}, 0)\}$. Alternatively, assignments can also be denoted as $A = \{x_1 = 0, x_7 = 1, x_{13} = 0\}$. Sometimes it is convenient to indicate that a variable x is assigned without specifying its actual value. In such cases, we will use the notation $\nu(x)$ to denote the binary value assigned to x. An assignment A is *complete* if $|A| = n$; otherwise it is *partial*. Evaluating a formula φ for a given truth assignment A yields three possible outcomes: $\varphi_{|A} = 1$ and we say that ω is

[1] This paper is organized in two main parts: the first one surveys well-established backtrack search SAT algorithms, whereas the second part addresses recent advances and current research directions.

satisfied and refer to A as a *satisfying assignment*; $\varphi_{|A} = 0$ in which case ω is unsatisfied and A is referred to as an *unsatisfying assignment*; and $\varphi_{|A} = X$ indicating that the value of ω cannot be resolved by the assignment. This last case can only happen when A is a partial assignment. An assignment partitions the clauses of ω into three sets: satisfied clauses (evaluating to 1); unsatisfied clauses (evaluating to 0); and unresolved clauses (evaluating to X). The unassigned literals of a clause are referred to as its *free literals*. A clause is said to be *unit* if the number of its free literals is one. The search process is declared to reach a *conflict* whenever $\varphi_{|A} = 0$ for a given assignment A.

Formula satisfiability is concerned with determining if a given formula ω is satisfiable and with identifying a satisfying assignment for it. Starting from an empty truth assignment, a backtrack search algorithm enumerates the space of truth assignments implicitly and organizes the search for a satisfying assignment by searching a *decision tree*. Each node in the decision tree specifies an elective assignment to an unassigned variable; such assignments are referred to as *decision assignments*. A *decision level* is associated with each decision assignment to denote its depth in the decision tree; the first decision assignment at the root of the tree is at decision level 1. The decision level at which a given variable x is either electively assigned or forcibly implied will be denoted by $\delta(x)$. When relevant to the context, the assignment notation introduced earlier may be extended to indicate the decision level at which the assignment occurred. Thus, $x = v@d$ would be read as "x becomes equal to v at decision level d".

Let the assignment of a variable x be implied due to a clause $\omega = (l_1 + \cdots + l_k)$ by using the unit clause rule [11]. The *antecedent assignment* of x, denoted as $A(x)$, is defined as the set of assignments to variables other than x with literals in ω. Intuitively, $A(x)$ designates those variable assignments that are directly responsible for implying the assignment of x due to ω. For example, given $\omega = (x + y + \neg z)$, the antecedent assignments of x, y and z are $A(x) = \{y = 0, z = 1\}$, $A(y) = \{x = 0, z = 1\}$, and $A(z) = \{x = 0, y = 0\}$, respectively. Note that the antecedent assignment of a decision variable is empty. In order to explain some of the concepts described in the remainder of the paper, we shall often analyze the sequences of implied assignments generated by Boolean Constraint Propagation (BCP) [48] (which is described in section 5.1). These sequences are captured by a directed acyclic *implication graph I* defined as follows:

1. Each vertex in I corresponds to a variable assignment $x = \nu(x)$.

2. The predecessors of vertex $x = \nu(x)$ in I are the antecedent assignments $A(x)$ corresponding to the unit clause ω that led to the implication of x. The directed edges from the vertices in $A(x)$ to vertex $x = \nu(x)$ are all labeled with ω. Vertices that have no predecessors correspond to decision assignments.

3. Special conflict vertices are added to I to indicate the occurrence of conflicts. The predecessors of a conflict vertex κ correspond to variable assignments that force a clause ω to become unsatisfied and are viewed as the antecedent assignment $A(\kappa)$. The directed edges from the vertices in $A(\kappa)$ to κ are all labeled with ω.

3. Backtrack search SAT algorithm

We start by describing a possible realization of a SAT algorithm which uses backtrack search, as illustrated in figure 1. The procedure `Preprocess` can either be used for algebraic simplifications, as in [18], or for deriving necessary assignments as in [29,45]. A generic organization of backtrack search for SAT is shown in figure 2.

```
//
// Global variables:  CNF formula φ
// Return value:  SATISFIABLE/UNSATISFIABLE
// Auxiliary variable:  Backtracking level β
//
SolveSatisfiability ()
{
  if (Preprocess (0) == CONFLICT) {
    return UNSATISFIABLE;
  }
  return BacktrackSearch (0,β);
}
```

Figure 1. Algorithm for satisfiability.

```
//
// Input argument:  Current decision level d
// Output argument:  Backtracking decision level β
// Return value:  SATISFIABLE or UNSATISFIABLE
//
BacktrackSearch (d, &β)
{
  if (Decide (d) != DECISION)
    return SATISFIABLE;
  while (TRUE) {
    if (Deduce (d) != CONFLICT) {
      if (BacktrackSearch (d + 1, β) == SATISFIABLE)
        return SATISFIABLE;
      else if (β != d || d == 0)
        Erase (d); return UNSATISFIABLE;
      }
    }
    if (Diagnose (d,β) == CONFLICT) {
      return UNSATISFIABLE;
    }
  }
}
```

Figure 2. Generic backtrack search algorithm.

The search is recursively organized in terms of the current *decision level*, d, denoting an elective decision assignment, and a backtracking decision level, that identifies the next decision assignment to be toggled. The backtrack algorithm is composed of three main engines:

- The decision engine (`Decide`) which selects a decision assignment each time it is called.
- The deduction engine (`Deduce`) which identifies assignments that are deemed necessary, given the current variable assignments and the most recent decision assignment, for satisfying the CNF formula.
- The diagnosis engine (`Diagnose`) which identifies the causes of a given conflicting partial assignment.

Besides the three engines, the backtrack search algorithm also invokes an `Erase` (d) procedure, that erases the variable assignments resulting from the most recent decision assignment. It is interesting to note that a significant number of backtrack search algorithms proposed by different authors can be cast as different configurations of the generic backtrack search algorithm, given suitable configurations of the three engines. For example, one possible configuration is the following:

- The decision engine randomly picks an unassigned variable x and assigns x value 1.
- The deduction engine implements Boolean Constraint Propagation (BCP) [48] and returns its outcome. A CONFLICT indication denotes the existence of unsatisfied clauses. Otherwise, in case of a NO_CONFLICT indication, the pure-literal rule [11] is applied.
- The diagnosis engine keeps track of which decision assignments have been toggled. Each time it is invoked, it checks whether at decision level d, the corresponding decision variable x has already been toggled. If not, it erases the variable assignments which are implied by the assignment on x, including the assignment on x, assigns the opposite value to x and returns a NO_CONFLICT indication. In contrast, if the value of x has already been toggled, it sets β to $d - 1$ and returns a CONFLICT indication.

From the above we can immediately conclude that such configuration represents one possible realization of the Davis–Putnam SAT algorithm [10,11]. Moreover, the generic algorithm can easily be customized to implement the POSIT [15] and the Tableau [9] SAT algorithms, among others.

There are three main approaches to improve backtrack search SAT algorithms. The first approach consists of fine-tuning the implementation details which, as shown by Freeman in [15], can be of key significance. For example, highly efficient implementations of BCP as well as of the `Erase` procedure introduce significant performance improvements over other less optimized realizations. The second approach for improving a SAT algorithm is the procedure for selecting decision assignments, which in most cases has a very significant impact on the overall efficiency of the algorithm. A large number of decision making heuristics for SAT have been proposed over the years, and a detailed account can be found for example in [15]. In addition, a brief overview of decision making heuristics will be given in section 4. Finally, the third approach for improving backtrack search algorithms consists of reducing the space that must actually be searched. Recent experimental results [3,35] strongly suggest that pruning the search can be extremely effective for many classes of instances of SAT. In the next section we review techniques commonly used for pruning the amount of search and illustrate how these techniques can be embedded in the proposed search framework. Furthermore, in section 6 we investigate other potentially useful pruning techniques, some of which have seldom been applied in the context of SAT.

4. Decision making heuristics

The heuristics used for variable selection during the search, and, consequently, the organization of the decision engine, represent a key aspect of backtrack search SAT algorithms [3,6,15,24,25,49]. Several heuristics have been proposed over the years, each denoting a tradeoff between computational requirements and the ability to reduce the amount of search [24]. Examples of decision making heuristics include Bohm's heuristic [6], the Jeroslow–Wang branching rule [25], MOM's heuristic [15] and BCP-based heuristics [3]. Most heuristics try to constrain the search as much as possible, by identifying at each step decision assignments that are expected to imply the largest number of variable assignments. For example, the one-sided Jeroslow–Wang heuristic [24,25] assigns value true to the literal that maximizes the following function:

$$J(L) = \sum_{L \in C_i} 2^{-n_i}, \qquad (1)$$

where n_i is the number of free literals in unresolved clause C_i. Hence, preference is given to satisfying a literal that occurs in the largest number of the smallest clauses. MOM's heuristic [15], for example, also gives preference to variables that occur in the smallest clauses, but variables are preferred if they simultaneously maximize their number of positive and negative literals in the smallest clauses. Bohm's heuristic [6] applies a similar reasoning, giving preference to variables that occur more often in the smallest clauses and, among these variables, to those for which both positive and negative literals occur in the smallest clauses. Other heuristics involve BCP-based probing of each unassigned variable, in order to deciding which variable will lead to the largest number of implied assignments [3,15]. For example, rel_sat [3] successfully applies this heuristic.

More recently, a different kind of decision heuristic has been proposed [36]. This new heuristic has been used in Chaff, a highly optimized SAT solver. More than to develop a well-behaviored heuristic, the motivation in Chaff has been to design a fast heuristic. In fact, one of the key properties of this strategy is the very low overhead, due to being independent of the variable state. As a result, the variable metrics are only updated when there is a conflict.

5. Pruning techniques for backtrack search

Practical search-based SAT algorithms incorporate a significant number of techniques for pruning the search. In the following sections we describe different techniques for pruning backtrack search.

5.1. Necessary assignments

The identification of the necessary assignments plays a key role in SAT and in Constraint Satisfaction, where it can be viewed as a form of reduction of the domain of each variable. In SAT algorithms, the most commonly used procedure for identifying necessary assignments is Boolean Constraint Propagation (BCP) [10,48]. The pseudo-code description of BCP is given in figure 3, and basically consists on the iterated application of the unit clause rule, originally described in [11]. Observe that one argument to the procedure is the decision level d, which in our framework is associated with every variable assignment.

From [48], we know that given a set of variable assignments, BCP identifies necessary assignments due to the unit clause rule in linear time on the number of literals of the CNF formula. However, BCP does not identify all necessary assignments given a set of variable assignments and a CNF formula. Consider, for example, the formula $(x_1 + x_2') \cdot (x_1 + x_2)$. For any assignment to variable x_2, variable x_1 must be assigned value 1 for preventing a conflict. Nevertheless, BCP applied to this CNF formula would not produce this straightforward conclusion.

One immediate extension to BCP are different forms of *value probing*. For example, for any value assignment to variable x_2, the assignment of x_1 to 1 is always implied. Thus, the value of x_1 *must* be assigned value 1. Different forms of value probing have

```
//
// Input argument:  Current decision level $d$
// Return value:  CONFLICT or NO_CONFLICT
//
Deduce ($d$)
{
  **while** (exists unit clause in $\varphi$) {
    Let $\omega$ be a unit clause with free literal $l$;
    Let $x$ be the variable associated with $l$;
    Assign $x$ so that $l = 1$ and $\omega$ becomes satisfied;
    **if** (exists unsatisfied clause) {
      **return** CONFLICT;
    }
  }
  **return** NO_CONFLICT;
}
```

Figure 3. Deduction engine implementing BCP.

been proposed over the years in different application domains (see, for example, [28]), but have seldom been incorporated in backtrack search algorithms for propositional satisfiability.

5.2. Clause recording

Clause recording is another pruning technique, which is tightly associated with non-chronological backtracking (described in the next section). This technique is often referred to as *nogood recording* in the literature on Truth Maintenance Systems [12,43] and Constraint Satisfaction Problems [39]. Basically, given a set of variable assignments, that is identified as representing a sufficient condition that leads to an identified conflict, clause recording consists in the creation of a new clause that prevents the same assignments from occurring simultaneously again during the subsequent search.

The pseudo-code for a diagnosis engine which implements non-chronological backtracking and clause recording is shown in figure 4. More complete details of creating clauses due to identified conflicts can be found in [3,35].

We illustrate clause recording with the example of figure 5. A subset of the CNF formula is shown, and we assume that the current decision level is 6, corresponding to the decision assignment $x_1 = 1$. As shown, this assignment yields a conflict involving clause ω_6. By inspection of the implication graph, we can readily conclude that a *sufficient condition* for this conflict to be identified is

$$(x_{10} = 0) \wedge (x_{11} = 0) \wedge (x_9 = 0) \wedge (x_1 = 1). \tag{2}$$

By creating clause $\omega_{10} = (x_{10} + x_{11} + x_9 + x_1')$ we prevent the same set of assignments from occurring again during the subsequent search.

Unrestricted clause recording is in most cases impractical. Recorded clauses consume memory and repeated recording of clauses eventually leads to the exhaustion of the available memory. Furthermore, large recorded clauses are known for not being particularly useful for search pruning purposes [35]. As a result, there are two main solutions

```
//
// Input argument:  Current decision level $d$
// Output argument:  Backtracking decision level $\beta$
// Return value:  CONFLICT or NO_CONFLICT
//
Diagnose ($d$, &$\beta$)
{
  $\omega_C$ = Create_Conflict_Induced_Clause();
  AddTo_CNF_Formula ($\omega_C$);
  $\beta$ = Compute_Max_Decision_Level ($\omega_C$);
  Erase ($d$);
  **return** (($\beta$ != $d$) ?  CONFLICT : NO_CONFLICT);
}
```

Figure 4. Outline of the diagnosis engine.

Current Truth Assignment: $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2, ...\}$
Current Decision Assignment: $\{x_1 = 1@6\}$

$\omega_1 = (x'_1 + x_2)$
$\omega_2 = (x'_1 + x_3 + x_9)$
$\omega_3 = (x'_2 + x'_3 + x_4)$
$\omega_4 = (x'_4 + x_5 + x_{10})$
$\omega_5 = (x'_4 + x_6 + x_{11})$
$\omega_6 = (x'_5 + x'_6)$
$\omega_7 = (x_1 + x_7 + x'_{12})$
$\omega_8 = (x_1 + x_8)$
$\omega_9 = (x'_7 + x'_8 + x'_{13})$
...

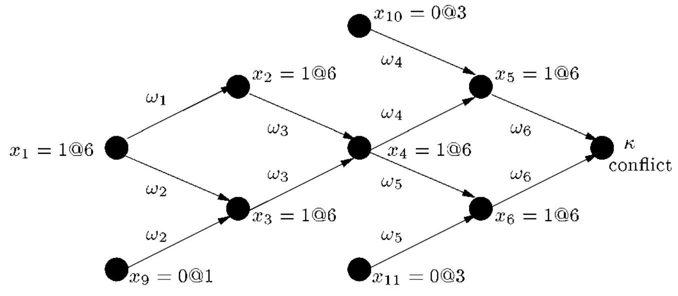

Figure 5. Example of conflict diagnosis with clause recording.

for clause recording. First, clauses can be temporarily recorded while they either imply variable assignments or are unit clauses, being discarded as soon as the number of unassigned literals is greater than one [3]. Second, clauses with a size less than a threshold k are kept during the subsequent search, whereas larger clauses are discarded as soon as the number of unassigned literals is greater than one [35]. We refer to this technique as *k-bounded learning*.

Finally, we note that clause recording can also be implemented when applying value probing techniques (described in section 5.1). This solution allows value probing to be used for deriving additional clauses, which further constrain the search. For example, for the clauses $(x_1 + x'_2) \cdot (x_1 + x_2)$, if we probe the assignment $x_1 = 0$, then applying BCP leads to a conflict. Diagnosis of this conflict [35] yields the unit clause (x_1), which immediately implies the assignment $x_1 = 1$.

5.3. *Non-chronological backtracking*

The chronological backtracking search strategy always causes the search to reconsider the last, yet untoggled, decision assignment. Chronological backtracking is the most often used strategy in SAT algorithms [2,4,9,10,14–16,27–29,37,45,48]. However, since this backtracking strategy uses no knowledge of the causes of conflicts, it can easily yield large sequences of conflicts *all* of which result from essentially the same variable assignments. In contrast, non-chronological backtracking strategies, originally proposed by Stallman and Sussman in [43] and further studied by Gaschnig [17] and others (see, for example [13]), attempt to identify the variable assignments causing a conflict and backtrack directly so that at least one of those variable assignments is modified. In the last couple of years a few SAT algorithms have been described in the literature which implement non-chronological backtracking [3,35]. In general recorded clauses are used for computing the backtracking decision level, which is defined as the highest decision level of all variable assignments of the literals in each newly recorded clause.

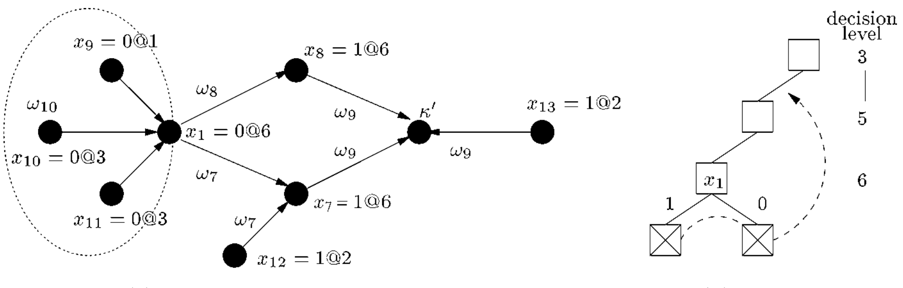

Figure 6. Computing the backtrack decision level.

In order to illustrate non-chronological backtracking, let us consider the example of figure 6, which continues the example in figure 5, after recording clause $\omega_{10} = (x_{10} + x_{11} + x_9 + x_1')$. At this stage BCP implies the assignment $x_1 = 0$ because clause ω_{10} becomes unit at decision level 6. By inspection of the CNF formula (see figure 5), we can conclude that clauses ω_7 and ω_8 imply the assignments shown, and so we obtain a conflict involving clause ω_9. It is straightforward to conclude that even though the current decision level is 6, all assignments directly involved in the conflict are associated with variables assigned at decision levels less than 6, the highest of which being 3. Hence we can backtrack immediately to decision level 3.

5.4. *Relevance-based learning*

A simple and highly effective improvement to clause recording is referred to as *relevance-based learning*, and was originally introduced by Bayardo and Schrag [3]. Suppose that we implement non-chronological backtracking with clause recording but, due to space restrictions, all recorded clauses must eventually be deleted. In general, recorded clauses can be deleted as soon as at least two literals become unassigned, since in this situation these clauses are not responsible for implying any variable assignments. Relevance-based learning basically consists of allowing recorded clauses to be deleted only when a larger number of literals becomes unassigned [3]. As a result, recorded clauses may get used again, either for yielding conflicts or for implying variable assignments. In contrast with unrestricted clause recording, the growth of the size of the CNF formula can be kept under tight control. For example, in [3] the number of allowed unassigned literals before deleting recorded clauses was either three and four.

From the current and the previous sections one can envision using k-bounded learning (described in section 5.2) with relevance-based learning. The search algorithm is organized so that all recorded clauses of size no greater than k are kept and larger clauses are deleted only after m literals have become unassigned. Given the experimental results obtained with the separate application of each of these techniques [3,35], their integration is expected to be particularly useful, given that only small clauses are added to the CNF formula, which in general introduce significant pruning, and the life span of larger clauses is increased.

5.5. Conflict-induced necessary assignments

The diagnosis of conflicts, as described in the previous sections, can be extended to further prune the amount of search. Let us consider again the implication graph of figure 5. By inspection we can conclude that the vertex $x_4 = 1$ is a *dominator* [46] of vertex $x_1 = 1$ with respect to vertex κ. As a result, the assignment $x_4 = 1$ leads by itself to the same conflict provided the assignments with decision levels less than 6 remain unchanged. Hence, instead of clause $\omega_{10} = (x_{10} + x_{11} + x_9 + x_1')$ we can create two clauses, $\omega_{11} = (x_{10} + x_{11} + x_4')$ and $\omega_{12} = (x_4 + x_9 + x_1')$, respectively. Consequently, by identifying dominators of the implication graph, we are able to reduce the size of recorded clauses [35]. Furthermore, the new clauses allow a larger number of assignments to be implied with BCP. For the two clauses above, ω_{11} implies the assignment $x_4 = 0$, whereas ω_{12} subsequently implies the assignment $x_1 = 0$. Observe that the original clause ω_{10} would not cause the implication of the assignment $x_4 = 0$. In general we refer to these extra implied assignments as conflict-induced necessary assignments.

6. Exploiting the structure of SAT instances

Besides the derivation of necessary assignments and the diagnosis of conflicts, other pruning techniques can be incorporated in SAT algorithms based on backtrack search. In this section we briefly review pruning techniques that exploit the structure of the CNF formula to simplify the search. These techniques have been extensively and successfully applied in solving different formulations of the set covering problem [7,8].

6.1. Clause subsumption

Consider the clauses $\omega_1 = (x_1 + x_2' + x_3' + x_4')$ and $\omega_2 = (x_1 + x_4')$. By inspection, we can readily conclude that $(\omega_2 = 1) \Rightarrow (\omega_1 = 1)$. Hence, we say that ω_2 *structurally subsumes* ω_1, and so ω_1 needs not be considered further for satisfiability purposes. During the search, variable assignments can naturally cause some clauses to become *dynamically* subsumed by others. For example, consider clauses $\omega_3 = (x_1 + x_2' + x_3' + x_4')$ and $\omega_4 = (x_1 + x_4' + x_5)$. Assuming that during the search x_5 is assigned value 0, then we say that ω_4 dynamically subsumes ω_3, and this holds as long as $x_5 = 0$. In SAT algorithms we can envision different implementations of clause subsumption:

1. Apply structural subsumption between every pair of clauses in the CNF formula prior to initiating the search.
2. After each decision assignment and associated implied assignments are identified, compute which clauses become dynamically subsumed by other clauses and remove them from the set of clauses. If these assignments must eventually be undone, then these clauses are no longer subsumed by other clauses.

3. Each time a conflict is diagnosed and a new clause is recorded, apply structural subsumption between the newly recorded clause and all other clauses in the CNF formula.

With respect to the above approaches for clause subsumption, case 1 produces a very small number of subsumed clauses [34] for most benchmarks. Case 2, though potentially promising, introduces significant computational overhead after each call to BCP, thus being impractical for highly efficient SAT algorithms. Finally, case 3 also incurs a significant computational overhead, as empirically shown in [34], and so it is hardly justifiable for practical instances of SAT.

6.2. *Formula partitioning*

In order to illustrate formula partitioning, let us consider the following CNF formula,

$$\begin{aligned}\varphi_1 = &\left(x_1' + x_3\right) \cdot \left(x_1' + x_2\right) \cdot \left(x_1 + x_2' + x_3'\right) \\ &\cdot \left(x_4 + x_5 + x_6'\right) \cdot \left(x_4' + x_6\right) \cdot \left(x_5' + x_6\right).\end{aligned} \tag{3}$$

For this CNF formula, the set of variables in clauses $\varphi_a = (x_1' + x_3) \cdot (x_1' + x_2) \cdot (x_1 + x_2' + x_3')$ is disjoint from the set of variables in clauses $\varphi_b = (x_4 + x_5 + x_6') \cdot (x_4' + x_6') \cdot (x_5' + x_6)$, and so the original CNF formula φ_1 can be *structurally partitioned* into the CNF sub-formulas φ_a and φ_b. It is straightforward to conclude that the two sub-formulas, φ_a and φ_b can be solved separately and any solution to φ_1 is composed of the set union of the solutions to each sub-formula. Assuming a CNF formula φ_1 with n variables that can be partitioned into m partitions each with n_j variables, we reduce the worst-case search space of $2^n = 2^{n_1} \times \cdots \times 2^{n_m}$ into a worst-case search space of $2^{n_1} + \cdots + 2^{n_m}$, thus introducing a significant reduction in the worst-case search space. Moreover, formula partitioning can be generalized to take into consideration variable assignments. For example, let us consider the following CNF sub-formula,

$$\begin{aligned}\varphi_2 = &\left(x_1' + x_3\right) \cdot \left(x_1' + x_2\right) \cdot \left(x_1 + x_2' + x_3'\right) \cdot \left(x_1 + x_7 + x_4'\right) \\ &\cdot \left(x_4' + x_5 + x_6'\right) \cdot \left(x_4' + x_6\right) \cdot \left(x_5' + x_6\right)\end{aligned} \tag{4}$$

and the assignment $x_7 = 1$ that satisfies clause $(x_1 + x_7 + x_4')$. As a result, we now obtain the CNF formula φ_1 given by (3) which, as we saw above, can be partitioned into the sub-formulas φ_a and φ_b. This form of CNF formula partitioning is referred to as *dynamic partitioning*, since it results from assignments made to the variables during the search. In SAT algorithms we can envision several implementations of clause partitioning.

1. Identify structural partitions before initiating the search.
2. After each decision assignment and associated implied assignments are identified, split the resulting CNF formula into dynamic partitions. If these assignments must eventually be undone, then the partitions no longer hold.

3. After each decision assignment and associated implied assignments are identified, implicitly select a partition and restrict the search to that partition. A partition is implicitly selected when decision assignments are restricted to variables in that partition.

In practical instances of SAT case 1 is not expected to be particularly relevant and case 2 incurs a significant computational overhead, making it hardly justifiable in practice [34]. Finally, the implicit identification of partitions, as empirically shown in [34] for the DIMACS benchmarks [26], can significantly simplify the search for different instances of SAT.

6.3. Partial solution caching

Formula partitioning can be further extended by allowing the solutions of partitions to be cached. This technique is commonly and effectively used in algorithms for solving covering problems [7,8]. Let us consider, for example, the CNF formula of (4) and the assignment $x_7 = 1$. As shown above, this causes φ_2 to be partitioned into the sub-formulas φ_a and φ_b. If we attempt to solve sub-formula φ_b, then one possible solution is $\{x_5 = 1, x_6 = 1\}$. As a result, during the subsequent search and each time this partition of φ_2 is created, by setting, for example, $x_7 = 1$, we can immediately say that a solution to the partition φ_b is $\{x_5 = 1, x_6 = 1\}$ *without* having to actually conduct a search to identify this solution. Consequently, we can create a database of triggering assignments and corresponding solution of a partition. These *precomputed* solutions can be used for preventing the search, for the solution of a given partition for which a solution has already been computed in the preceding search. For our example, an entry in such a database would be $\langle\{x_7 = 1\}, \{x_5 = 1, x_6 = 1\}\rangle$, which basically states that in the presence of the assignment $x_7 = 1$, we can readily apply the assignments $\{x_5 = 1, x_6 = 1\}$, which represent the solution of partition φ_b of the original CNF formula. If the objective assignments are not consistent with already made assignments (e.g., by having, for example, $x_5 = 0$) then the cached solution is simply not used. As shown in [7] partial solution caching can introduce improvements to the running times of orders of magnitude in standard benchmarks.

7. Experimental evidence

In this section we illustrate the potential practical application of backtrack search SAT algorithms. For this purpose, we evaluated two well-known backtrack search SAT algorithms, ntab [9] and POSIT [15], which are based on chronological backtracking search strategies, and incorporate highly effective variable selection heuristics. In addition, three recent backtrack search SAT algorithms, that implement most of the techniques described in this paper, SATO [49], rel_sat [3] and GRASP [35] were evaluated. Finally, a general purpose constraint solver, wcsat [47], was also evaluated. The experiment consisted in running the different algorithms on several real-world practical instances from circuit verification. All instances are unsatisfiable. (We note that this is

Table 1
Results on benchmark examples

Example	SAT	ntab	POSIT	wcsat	SATO	rel_sat	GRASP
bench1	N	–	–	12.75	0.48	2.37	48.70
bench2	N	–	–	26.37	1.09	1.66	16.92
bench3	N	–	–	17.71	0.96	1.39	13.22
bench4	N	–	–	18.68	0.77	1.72	13.19
bench5	N	–	–	21.80	1.46	1.71	17.82
bench6	N	–	–	–	–	146.09	198.93
bench7	N	–	–	–	–	101.24	154.04
bench8	N	–	–	–	–	58.40	185.99
bench9	N	–	–	–	–	24.37	165.36
bench10	N	–	–	–	–	–	111.70
bench11	N	–	–	–	–	–	223.80
bench12	N	–	–	–	–	80.81	93.83

true in general for circuit verification.) The results are shown in table 1.[2] Entries marked with '–' indicate that the algorithm did not finish in 300 seconds of allowed CPU time. As can be readily concluded, backtrack search SAT algorithms that implement the techniques described in this paper are by far the most competitive in solving these particular types of instances. Furthermore, we observe that the different organizations of rel_sat [3] and of GRASP [34,35] lead to somewhat different results. rel_sat is in general faster, but may be unable to solve a few instances. The same holds true with SATO. On the other hand, GRASP is slower in most benchmarks but, for the examples shown, it is more robust since it does not quit for any instance. Finally, we observe that the version of GRASP used above implements all the techniques described in section 5, including relevance-based learning as described in [3]. The above results indicate that backtrack search SAT algorithms, specifically those that implement different search pruning techniques, can be the only algorithmic solution for successfully solving specific classes of practical instances on SAT. These results further substantiate further work on developing new and more effective pruning techniques for backtrack search SAT algorithms.

8. Recent work

In this section we describe recent work on backtrack search SAT algorithms, namely simplification techniques, probing methods for the identification of necessary assignments, randomization and restarts, and implementation issues.

8.1. Advanced techniques

Recent work on improving backtrack satisfiability algorithms has included advanced techniques, such as the simplification of a given formula [30,33] and the identification of necessary assignments [28,32,44].

[2] The results of GRASP and SATO were obtained on a Pentium 200 MHz machine. The other results were obtained on a Pentium Pro 200 MHz machine.

Resolution is probably the most well-known technique among the existing simplification techniques. Resolution can be regarded as a general technique for deriving new clauses. For example, given clauses $(x + y + z')$ and $(x' + y + z')$, resolution allows deriving clause $(y+z')$. However, when applying general resolution, we may generate in the worst case an exponential number of clauses, many of which are redundant and often irrelevant. As a consequence of general resolution being computationally too expensive, some alternative techniques have been developed. These techniques indirectly identify which resolution operations to apply [22].

Two-variable equivalence is another formula simplification procedure. Let us consider an example, using the pair of clauses $(x_1 + x_2')$ and $(x_1' + x_2)$. For any assignment that satisfies the two clauses, the truth values of x_1 and x_2 must be the same. For this reason, variables x_1 and x_2 are said to be equivalent. Therefore one can replace x_2 by x_1 on all occurrences of x_2 (or vice-versa).

Recent work on formula simplification also includes *equivalency reasoning* [30]. The main contribution of equivalency reasoning is to establish conditions under which the CNF formula can be simplified by identifying pairs of equivalent variables and by, subsequently, removing one variable from each such pair of variables.

Additional work on formula simplification consists of identifying patterns in CNF formulas from which new unit and binary clauses can be inferred [33]. The objective of studying clause patterns is to indirectly select to which sets of clauses the resolution operation should actually be applied to. In practice, the inference of clauses contributes to adding more information to the problem specification, and therefore can potentially simplify the subsequent search. Moreover, we should note that the inference of binary clauses can contribute to finding more equivalent variables.

As mentioned above, other advanced techniques include probing for identifying necessary assignments. From section 5.1, we know that BCP does not identify all necessary assignments given a set of variable assignments and a CNF formula. Moreover, the backtrack search procedure can and has been augmented with different probing techniques, namely by using *Recursive Learning* [28,32] and by applying the branch-merge rule of *Stålmark's Method* [22,44]. Recursive Learning and Stålmark's Method can be respectively interpreted as concrete forms of *clause probing* and *variable probing*.

Clause probing [28,32] consists of the recursive evaluation of clause satisfiability requirements for identifying common assignments to variables. Common assignments are deemed necessary for a given clause to become satisfiable and consequently for the instance of SAT to be satisfiable. Let us consider the formula $\varphi_a = (x_1 + x_2) \cdot (x_1' + x_3) \cdot (x_2' + x_3)$. Clause probing would try to assign $x_1 = 1$ and then $x_2 = 1$ to satisfy clause $(x_1 + x_2)$, and in each case it would propagate the implied assignments. For this example observe that every way of satisfying clause $(x_1 + x_2)$ implies $x_3 = 1$. Hence, $x_3 = 1$ is a necessary assignment.

The main goal of variable probing is to identify common assignments to variables, detecting and merging equivalent branches in the search tree. In other words, variable probing is defined by the recursive application of the branch-merge rule to each variable [22]. Let us now consider the formula $\varphi_b = (x_1 + x_2) \cdot (x_1' + x_3) \cdot (x_2' + x_4) \cdot (x_3' + x_4)$. We will start by trying variable probing with variable x_1. We conclude that both assignments, i.e., $x_1 = 0$ and $x_1 = 1$, imply $x_4 = 1$. Hence, $x_4 = 1$ is a necessary assignment.

Note that both clause and variable probing can be applied using recursion of arbitrary depth, respectively for clauses and for variables. Moreover, both approaches are based on similar reasoning, despite one being based on clauses and the other on variables. More interestingly, observe that these methods can be integrated into backtrack search SAT algorithms and can also be used together.

8.2. *Search strategies*

The utilization of different forms of randomization in backtrack search SAT algorithms has also seen increasing acceptance in recent years [21]. Randomization consists of introducing a certain degree of uncertainty in selecting branching variables and values during the search. On the other hand, restarts allow repeatedly restarting the search each time a given limit number of decisions is reached. Moreover, restarts with randomization allow searching different regions of the search space and have been shown to yield dramatic improvements on satisfiable instances that exhibit heavy-tailed behavior [1,20,36].

8.3. *Fast implementations*

Implementation issues for SAT solvers include the design of suitable data structures for storing variables, clauses and literals. The elected data structures dictate the way BCP and conflict analysis are implemented and have significant impact on the run time performance of the SAT solver. The simplest and most widely used representation a formula consists of having a set of literals for each variable and for each clause [3,9,15,35]. The state of each clause is determined by a set of counters.

A variation to this data structure was proposed in SATO [49]. In SATO, each clause has two additional pointers: *head* and *tail* to point to its first and last unassigned literals, which can be updated either during BCP or during backtracking. Moreover, each variable only maintains its own head and tail lists, i.e., its occurrences as respectively the first and the last unassigned literals in a clause. Consequently, instead of keeping counters for each clause, unit clauses and conflict clauses are detected by examining the relative positions of the head and tail pointers of each clause.

More recently, Chaff [36] has achieved significant performance gains through a careful engineering of all aspects of the search, especially a low overhead decision strategy (see section 4) and a particularly efficient implementation of BCP. Chaff's implementation is different from SATO's in that it does not require a fixed direction of motion for the two additional pointers, in Chaff referred to as *watch pointers*. In SATO, the head literal can only move towards the tail literal and vice versa, and therefore variable unassignments have the same overhead as variable assignments. In contrast, in Chaff pointers do not need to be updated during backtracking, and so only variable state needs to be recovered.

8.4. Multiprocessor implementations

Another recent research direction in backtrack search SAT algorithms are multiprocessor approaches. Different parallelization solutions can be envisioned. One example is to partition the problem instance into several parts that are run by the same algorithm on different processors. Another example is to partition the search space, by ensuring that different processors analyze different portions of the search space.

The first parallel procedure was proposed by Böhm and Speckenmeyer [5]. These authors divided the input formula into subformulas which are distributed among processors. Another well-known multiprocessor implementation is PSATO [50], a master–slave distributed SAT solver, where each slave process executes a modified version of the sequential SAT solver SATO. The master process distributes the given problem among the slaves in such a way that the slaves explore non-overlapping portions of the search space in discrete time segments. In this way, PSATO is able to exploit parallelism without incurring the overhead of redundant search, i.e., parallel processes searching the same portion of the search space. More recently, the organization of PSATO has been extended in PaSAT [42] that integrates the exchange of recorded clauses between concurrent processes. The recorded clauses that become exchanged are selected using clause length as a simple criterion.

Another technique which can take advantage of multiprocessor implementations is algorithm portfolio design [19]. This approach is motivated by the fact that different algorithms give rise to different probability distributions. In this context, Gomes and Selman [19] have proposed to take advantage of such differences by combining several algorithms into a portfolio. Moreover, the different algorithms can be executed in parallel.

9. Conclusions

Backtrack search SAT algorithms are the option of choice for several classes of instances of SAT and whenever the objective is to prove unsatisfiability. Recent years have seen significant contributions for improving the efficiency of backtrack search SAT algorithms, that involve in most cases different techniques for pruning the search. Extensive experimental evaluation of these techniques [3,34,35,49] shows dramatic improvements, for a large number of classes of instances of SAT, over less optimized backtrack search SAT algorithms, in particular the Davis–Putnam procedure [10] and several of its improvements [15,31]. Despite the aforementioned contributions, we believe further experimental evaluation is required. First, because an empirical categorization of the different techniques might provide useful insights. Second, because a unified algorithmic framework, derived from these empirical insights, might allow solving a larger number of instances of SAT.

Acknowledgements

The authors would like to thank Thomas Glass, from Siemens AG, for providing the real-world instances of SAT used in the paper, as well as John Franco for his insightful comments, which helped improving this paper.

References

[1] L. Baptista and J.P. Marques-Silva, Using randomization and learning to solve hard real-world instances of satisfiability, in: *International Conference on Principles and Practice of Constraint Programming* (September 2000) pp. 489–494.

[2] P. Barth, A Davis–Putnam enumeration procedure for linear pseudo-Boolean optimization, Technical Report MPI-I-2-003, MPI (January 1995).

[3] R. Bayardo Jr. and R. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: *Proceedings of the National Conference on Artificial Intelligence* (1997) pp. 203–208.

[4] C.E. Blair, R.G. Jeroslow and J.K. Lowe, Some results and experiments in programming techniques for propositional logic, Computers and Operations Research 13(5) (1986) 633–645.

[5] M. Böhm and E. Speckenmeyer, A fast parallel SAT-solver – efficient workload balancing, in: *Third International Symposium on Artificial Intelligence and Mathematics* (1994).

[6] M. Buro and H. Kleine-Büning, Report on a SAT competition, Technical Report, University of Paderborn (November 1992).

[7] O. Coudert, On solving covering problems, in: *Proceedings of the ACM/IEEE Design Automation Conference* (June 1996) pp. 197–202.

[8] O. Coudert and J.C. Madre, New ideas for solving covering problems, in: *Proceedings of the ACM/IEEE Design Automation Conference* (June 1995).

[9] J. Crawford and L. Auton, Experimental results on the cross-over point in satisfiability problems, in: *Proceedings of the National Conference on Artificial Intelligence* (1993) pp. 22–28.

[10] M. Davis, G. Logemann and D. Loveland, A machine program for theorem-proving, Communications of the ACM 5 (July 1962) 394–397.

[11] M. Davis and H. Putnam, A computing procedure for quantification theory, Journal of the ACM 7 (1960) 201–215.

[12] J. de Kleer, An assumption-based TMS, Artificial Intelligence 28 (1986) 127–162.

[13] R. Dechter, Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition, Artificial Intelligence 41 (1989/1990) 273–312.

[14] O. Dubois, P. Andre, Y. Boufkhad and J. Carlier, SAT versus UNSAT, in: *Second DIMACS Implementation Challenge*, eds. D.S. Johnson and M.A. Trick (American Mathematical Society, 1993).

[15] J.W. Freeman, Improvements to propositional satisfiability search algorithms, PhD thesis, University of Pennsylvania, Philadelphia, PA (May 1995).

[16] G. Gallo and G. Urbani, Algorithms for testing the satisfiability of propositional formulae, Journal of Logic Programming 7 (1989) 45–61.

[17] J. Gaschnig, Performance measurement and analysis of certain search algorithms, PhD thesis, Carnegie-Mellon University, Pittsburgh, PA (May 1979).

[18] A.V. Gelder and Y.K. Tsuji, Satisfiability testing with more reasoning and less guessing, in: *Second DIMACS Implementation Challenge*, eds. D.S. Johnson and M.A. Trick (American Mathematical Society, 1993).

[19] C.P. Gomes and B. Selman, Algorithm portfolio design: theory vs. practice. in: *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence* (1997).

[20] C.P. Gomes, B. Selman, N. Crato and H. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, Journal of Automated Reasoning 24(1/2) (2000) 67–100.

[21] C.P. Gomes, B. Selman and H. Kautz, Boosting combinatorial search through randomization, in: *Proceedings of the National Conference on Artificial Intelligence* (July 1998).

[22] J.F. Groote and J.P. Warners, The propositional formula checker Heerhugo, in: *SAT 2000*, eds. I. Gent, H. van Maaren and T. Walsh (IOS Press, 2000) pp. 261–281.

[23] J. Franco, J. Gu, P.W. Purdom and B.W. Wah, Algorithms for the satisfiability (SAT) problem: A survey, in: *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (American Mathematical Society, 1997) pp. 19–152.

[24] J.N. Hooker and V. Vinay, Branching rules for satisfiability, Journal of Automated Reasoning 15 (1995) 359–383.

[25] R.G. Jeroslow and J. Wang, Solving propositional satisfiability problems, Annals of Mathematics and Artificial Intelligence 1 (1990) 167–187.

[26] D.S. Johnson and M.A. Trick (eds.), *Second DIMACS Implementation Challenge* (American Mathematical Society, 1993). DIMACS SAT instances available from URL ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf.

[27] S. Kim and H. Zhang, ModGen: Theorem proving by model generation, in: *Proceedings of the National Conference on Artificial Intelligence* (1994) pp. 162–167.

[28] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks* (Kluwer Academic, 1997).

[29] T. Larrabee, Test pattern generation using Boolean satisfiability, IEEE Transactions on Computer-Aided Design 11(1) (1992) 4–15.

[30] C.-M. Li, Integrating equivalency reasoning into Davis–Putnam procedure, in: *Proceedings of the National Conference on Artificial Intelligence* (July 2000) pp. 291–296.

[31] C.-M. Li and Anbulagan, Look-ahead versus look-back for satisfiability problems, in: *International Conference on Principles and Practice of Constraint Programming* (1997).

[32] J.P. Marques-Silva, Improving satisfiability algorithms by using recursive learning, in: *Proceedings of the International Workshop on Boolean Problems* (September 1998) pp. 47–58.

[33] J.P. Marques-Silva, Algebraic simplification techniques for propositional satisfiability, in: *International Conference on Principles and Practice of Constraint Programming* (September 2000) pp. 537–542.

[34] J.P. Marques-Silva and A.L. Oliveira, Improving satisfiability algorithms with dominance and partitioning, in: *Proceedings of the ACM/IEEE International Workshop on Logic Synthesis* (May 1997).

[35] J.P. Marques-Silva and K.A. Sakallah, GRASP: A new search algorithm for satisfiability, in: *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design* (November 1996) pp. 220–227.

[36] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, Engineering an efficient SAT solver, in: *Proceedings of the Design Automation Conference* (2001).

[37] D. Pretolani, Efficiency and stability of hypergraph SAT algorithms, in: *Second DIMACS Implementation Challenge*, eds. D.S. Johnson and M.A. Trick (American Mathematical Society, 1993).

[38] A. Saldanha and V. Singhal, Solving satisfiability in CAD problems, in: *Proceedings of the Cadence Technical Conference* (May 1997) pp. 191–195.

[39] T. Schiex and G. Verfaillie, Nogood recording for static and dynamic constraint satisfaction problems, in: *Proceedings of the International Conference on Tools with Artificial Intelligence* (1993) pp. 48–55.

[40] B. Selman and H. Kautz, Domain-independent extensions to GSAT: Solving large structured satisfiability problems, in: *Proceedings of the International Joint Conference on Artificial Intelligence* (1993) pp. 290–295.

[41] B. Selman, H. Levesque and D. Mitchell, A new method for solving hard satisfiability problems, in: *Proceedings of the National Conference on Artificial Intelligence* (1992) pp. 440–446.

[42] C. Sinz, W. Blochinger and W. Küchlin, PaSAT: Parallel SAT-checking with lemma exchange: implementation and applications, in: *LICS Workshop on Theory and Applications of Satisfiability Testing*, Electronic Notes in Discrete Mathematics (Elsevier Science, June 2001).

[43] R.M. Stallman and G.J. Sussman, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, Artificial Intelligence 9 (October 1977) 135–196.

[44] G. Stålmarck, A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula (1989). Swedish Patent 467 076 (approved 1992); US Patent 5 276 897 (approved 1994); European Patent 0 403 454 (approved 1995).

[45] P.R. Stephan, R.K. Brayton and A.L. Sangiovanni-Vincentelli, Combinational test generation using satisfiability, IEEE Transactions on Computer-Aided Design 15(9) (September 1996) 1167–1176.

[46] R.E. Tarjan, Finding dominators in directed graphs, SIAM Journal on Computing 3(1) (1974) 62–89.

[47] M. Yokoo, Asynchronous weak-commitment search for solving distributed constraint satisfaction problems, in: *Proceedings of International Conference on Principles and Practice of Constraint Programming* (1995).

[48] R. Zabih and D.A. McAllester, A rearrangement search strategy for determining propositional satisfiability, in: *Proceedings of the National Conference on Artificial Intelligence* (1988) pp. 155–160.

[49] H. Zhang, SATO: An efficient propositional prover, in: *Proceedings of the International Conference on Automated Deduction* (July 1997) pp. 272–275.

[50] H. Zhang, M.P. Bonacina and J. Hsiang, PSATO: a distributed propositional prover and its application to quasigroup problems, Journal of Symbolic Computation (1996) 1–18.