An Overview of Backtrack Search Satisfiability Algorithms

João P. Marques Silva

Cadence European Laboratories
IST/INESC
1000 Lisboa, Portugal
Tel: 351-1-3100387
e-mail: jpms@inesc.pt

Abstract

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.

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. Local search algorithms [23, 24] 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 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 [2, 25, 26]. 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 [25]. Moreover, even though not competitive in practice, procedures based on different forms of algebraic manipulation have been used as preprocessing techniques in several recently proposed SAT algorithms [11, 17].

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. Consequently, we propose in this paper to overview backtrack search algorithms for SAT, giving a particular emphasis to the techniques that are commonly used for pruning the search as well as indicating 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. Afterwards, in Section 3, we describe SAT algorithms based on backtrack search. Techniques commonly used for pruning the amount of search are reviewed in Section 4, whereas other less well-known techniques are described in Section 5. Experimental evidence of the practical application of backtrack search SAT algorithms is given Section 6. Finally, Section 7 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 φ 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 φ 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 φ is satisfied and refer to A as a *satisfying assignment*; $\varphi|_A = 0$ in which case φ is unsatisfied and A is referred to as an *unsatisfying assignment*; and $\varphi|_A = X$ indicating that the value of φ cannot be resolved by the assignment. This last case can only happen when A is a partial assignment. An assignment partitions the clauses of φ 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 φ 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 maintaining 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 $\omega = (l_1 + \ldots + l_k)$ by using the unit clause rule [7]. 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) [32] (which is described in Section 4.1). 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 [11], or for deriving necessary assignments as in [19, 29]. A generic organization of backtrack search for SAT is shown in Figure 2. 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:

```
//
// 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 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) [32] 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 [7] 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 [7, 8]. Moreover, the generic algorithm can easily be customized to implement the POSIT [12] and the Tableau [6] 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 J. Freeman in [12], 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 in [12]. Finally, the third approach for improving backtrack search algorithms consists in reducing the space that must actually be searched. Recent experimental results [2, 26] 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 5 we investigate other potentially useful pruning techniques, some of which have seldom been applied in the context of SAT.

4 Pruning Techniques for Backtrack Search

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

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

Figure 3: Deduction engine implementing BCP

4.1 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) [32]. 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 [7]. Observe that one argument to the procedure is the decision level d, which in our framework is associated with every variable assignment.

From [32], 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 been proposed over the years in different application domains (see for example [13, 18]), but have seldom been incorporated in algorithms for propositional satisfiability. Nevertheless, for satisfiability problems in boolean networks, a domain closely related to propositional satisfiability [19], a form of value probing referred to as *recursive learning* is commonly used [18].

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

Figure 4: Outline of the diagnosis engine

4.2 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 [10, 28] and Constraint Satisfaction Problems [22]. 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 [2, 26].

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

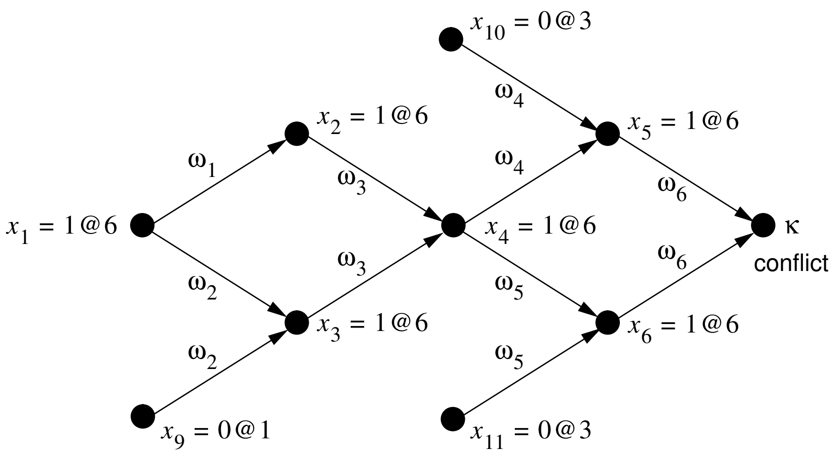
Unrestricted clause recording is in most cases impractical. Recorded clauses consume memory and repeated recording of clauses eventually leads to the exhaustion of the available memory. Furthermore, large recorded clauses are known for not being particularly useful for search pruning purposes [26]. As a result, there are two main solutions 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 [2]. Second, clauses with a size less than a threshold k are kept

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

Current Decision Assignment: $\{x_1 = 1@6\}$

$$\omega_1 = (x_1' + x_2)$$
$$\omega_2 = (x_1' + x_3 + x_9)$$
$$\omega_3 = (x_2' + x_3' + x_4)$$
$$\omega_4 = (x_4' + x_5 + x_{10})$$
$$\omega_5 = (x_4' + x_6 + x_{11})$$
$$\omega_6 = (x_5' + x_6')$$
$$\omega_7 = (x_1 + x_7 + x_{12}')$$
$$\omega_8 = (x_1 + x_8)$$
$$\omega_9 = (x_7' + x_8' + x_{13}')$$
$$\ldots$$

CNF Formula



Implication Graph for Current Decision Assignment

Figure 5: Example of conflict diagnosis with clause recording

during the subsequent search, whereas larger clauses are discarded as soon as the number of unassigned literals is greater than one [26]. 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 4.1). 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 [26] yields the unit clause (x_1), which immediately implies the assignment $x_1 = 1$.

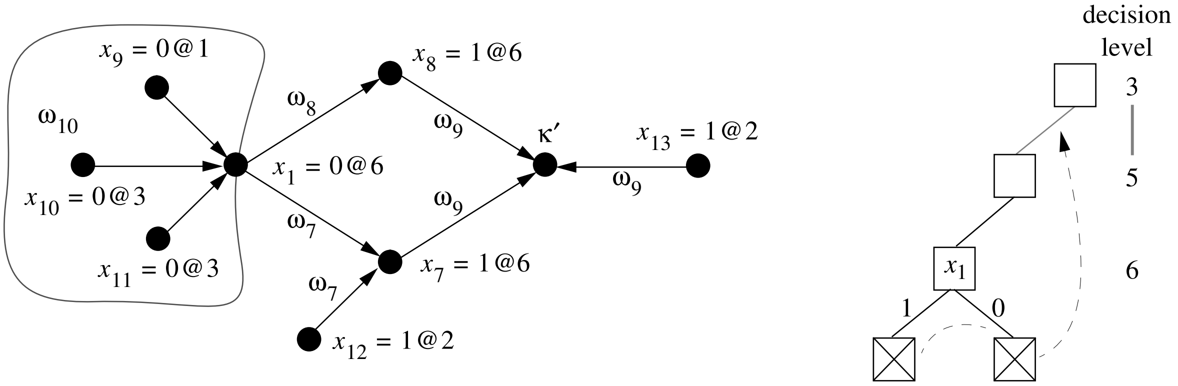
4.3 Non-Chronological Backtracking

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 [1, 3, 6, 8, 11, 12, 14, 17-19, 21, 29, 32]. 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 [28] and further studied by J. Gaschnig [15] and others (see for example [9]), 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 [2, 26]. 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.

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 $x_1 = 1$ 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.

4.4 Relevance-Based Learning

A simple and highly effective improvement to clause recording is referred to as *relevance-based learning*, and was originally introduced by R. Bayardo and R. Schrag in [2]. Suppose that we implement non-chronological backtracking with clause recording but, due to space restrictions, all



(a) Conflicting implication sequence (b) Decision tree

Figure 6: Computing the backtracking decision level

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 [2]. 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 [2] 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 4.2) 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 [2, 26], 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.

4.5 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* [30] 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 [26]. 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.

5 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 [4, 5].

5.1 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 are 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 for most benchmarks a very small number of subsumed clauses [27]. 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 in significant computational overhead, as empirically shown in [27], and so it is hardly justifiable for practical instances of SAT.

5.2 Formula Partitioning

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

$$\begin{aligned} \varphi_1 = {} & (x_1' + x_3) \cdot (x_1' + x_2) \cdot (x_1 + x_2' + x_3') \cdot \\ & (x_4 + x_5 + x_6') \cdot (x_4' + x_6) \cdot (x_5' + x_6) \end{aligned} \tag{1}$$

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 with n variables that can be partitioned into m partitions each with n_j variables, we reduce the worst-case search space of $2^n = 2^{n_1} \times \ldots \times 2^{n_m}$ into a worst-case search space of $2^{n_1} + \ldots + 2^{n_m}$, 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,

$$\begin{aligned} \varphi_2 = {} & (x_1' + x_3) \cdot (x_1' + x_2) \cdot (x_1 + x_2' + x_3') \cdot \\ & (x_1 + x_7 + x_4') \cdot (x_4 + x_5 + x_6') \cdot \\ & (x_4' + x_6) \cdot (x_5' + x_6) \end{aligned} \tag{2}$$

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 (1) 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 in significant computational overhead, making it hardly justifiable in practice [27]. Finally, the implicit identification of partitions, as empirically shown in [27] for the DIMACS benchmarks [16], can significantly simplify the search for different instances of SAT.

5.3 Partial Solution Caching

Formula partitioning can be further extended by allowing the solution of partitions to be cached. This technique is commonly and effectively used in algorithms for solving covering problems [4, 5]. Let us consider for example the CNF formula of (2) 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

Table 1: Results on benchmark examples[a]

Example	SAT	ntab	posit	wcsat	rel_sat	grasp
bench1	N	—	—	12.75	2.37	48.70
bench2	N	—	—	26.37	1.66	16.92
bench3	N	—	—	17.71	1.39	13.22
bench4	N	—	—	18.68	1.72	13.19
bench5	N	—	—	21.80	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

a. The results of GRASP were obtained on a Pentium 200 MHz machine. The other results were obtained on a Pentium Pro 200 MHz machine.

solution has already been computed in the preceding search. For our example, an entry in such 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 [4] partial solution caching can introduce improvements to the running times of orders of magnitude in standard benchmarks.

6 Experiments

In this section we illustrate the potential practical application of recent backtrack search SAT. Table 1 includes results of different SAT algorithms (ntab [6], POSIT [12], wcsat [31], rel_sat [2] and GRASP [26]) on real-world practical instances from circuit verification. All instances are unsatisfiable. (We note that this is true in general for circuit verification.) 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 [2] and of GRASP [25-27] lead to somewhat different results. rel_sat is in general faster, but may be unable to solve a few instances. 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 4, including relevance-based learning as described in [2].

The above results allow us to conclude 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.

7 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 [2, 26, 27] 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 [8] and several of its improvements.

Despite the aforementioned contributions, 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.

Acknowledgments

The author would like to thank Thomas Glass, from Siemens AG, for providing the real-world instances of SAT used in the paper.

References

[1] P. Barth, "A Davis-Putnam Based Enumeration Algorithm for Linear pseudo-Boolean Optimization," Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, 1995.

[2] R. Bayardo Jr. and R. Schrag, "Using CSP Look-Back Techniques to Solve Real-World SAT Instances," in *Pro-*

ceedings of the National Conference on Artificial Intelligence (AAAI-97), 1997. (source code for rel_sat available from http://www.cs.utexas.edu/users/bayardo/bin/rel_sat.tar.Z.)

[3] C. E. Blair, R. G. Jeroslow and J. K. Lowe, "Some Results and Experiments in Programming Techniques for Propositional Logic," *Computers and Operations Research*, vol. 13, no. 5, pp. 633-645, 1986.

[4] O. Coudert, "On Solving Covering Problems," in *Proceedings of the Design Automation Conference*, June 1996.

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

[6] J. Crawford and L. Auton, "Experimental Results on the Cross-Over Point in Satisfiability Problems," in *Proc. National Conference on Artificial Intelligence (AAAI-93)*, pp. 22-28, 1993. (source code for NTAB available from http://www.cirl.uoregon.edu/crawford/ntab.tar.)

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

[8] M. Davis, G. Logemann and D. Loveland, "A Machine Program for Theorem-Proving," *Communications of the ACM*, vol. 5, pp. 394-397, July 1962.

[9] R. Dechter, "Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition," *Artificial Intelligence*, vol. 41, pp. 273-312, 1989/90.

[10] J. de Kleer, "An Assumption-Based TMS," *Artificial Intelligence*, vol. 28, pp. 127-162, 1986.

[11] O. Dubois, P. Andre, Y. Boufkhad and J. Carlier, "SAT versus UNSAT," *Second DIMACS Implementation Challenge*, D. S. Johnson and M. A. Trick (eds.), 1993.

[12] J. W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms*, Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, May 1995. (source code for POSIT available from ftp://ftp.cis.upenn.edu/pub/freeman/posit-1.0.tar.gz.)

[13] E. C. Freuder, "Synthesizing Constraint Expressions," *Communications of the ACM*, vol. 21, no. 11, pp. 958-966, November 1978.

[14] G. Gallo and G. Urbani, "Algorithms for Testing the Satisfiability of Propositional Formulae," *Journal of Logic Programming*, vol. 7, pp. 45-61, 1989.

[15] J. Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms*, Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, CMU-CS-79-124, May 1979.

[16] D. S. Johnson and M. A. Trick (eds.), *Second DIMACS Implementation Challenge*, 1993. (DIMACS benchmarks available from ftp://Dimacs.Rutgers.EDU/pub/challenge/sat/benchmarks/cnf. UCSC benchmarks available in /pub/challenge/sat/contributed/UCSC.)

[17] S. Kim and H. Zhang, "ModGen: Theorem proving by model generation," in *Proc. National Conference of American Association on Artificial Intelligence (AAAI-94)*, pp. 162-167, 1994.

[18] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks*, Kluwer Academic Publishers, 1997.

[19] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 4-15, January 1992.

[20] D. A. McAllester, "An Outlook on Truth Maintenance," AI Memo 551, MIT AI Laboratory, August 1980.

[21] D. Pretolani, "Efficiency and Stability of Hypergraph SAT Algorithms," *Second DIMACS Implementation Challenge*, D. S. Johnson and M. A. Trick (eds.), 1993.

[22] 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*, pp. 48-55, 1993.

[23] B. Selman, H. Levesque and D. Mitchell, "A New Method for Solving Hard Satisfiability Problems," in *Proceedings of the National Conference on Artificial Intelligence (AAAI-92)*, pp. 440-446, 1992.

[24] 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*, pp. 290-295, 1993.

[25] J. P. M. Silva, *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of Michigan, May 1995.

[26] J. P. M. Silva and K. A. Sakallah, "GRASP—A New Search Algorithm for Satisfiability," in *Proceedings of the International Conference on Computer-Aided Design*, November 1996. (source code for GRASP available from http://algos.inesc.pt/pub/users/jpms/soft/grasp/grasp.tar.gz.)

[27] J. P. M. Silva and A. L. Oliveira, "Improving Satisfiability Algorithms with Dominance and Partitioning," in *International Workshop on Logic Synthesis*, May 1997.

[28] R. M. Stallman and G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence*, vol. 9, pp. 135-196, October 1977.

[29] P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 15, no. 9, pp. 1167-1176, September 1996.

[30] R. E. Tarjan, "Finding Dominators in Directed Graphs," *SIAM Journal on Computing*, vol. 3, no. 1, pp. 62-89, March 1974.

[31] M. Yokoo, "Asynchronous Weak-Commitment Search for Solving Distributed Constraint Satisfaction Problems," in *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, LNCS 976, 1995.

[32] R. Zabih and D. A. McAllester, "A Rearrangement Search Strategy for Determining Propositional Satisfiability," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 155-160, 1988.