

# The Quest for Efficient Boolean Satisfiability Solvers



Lintao Zhang, Sharad Malik

Department of Electrical Engineering, Princeton University, Princeton, NJ 08544  
{lintaoz, sharad}@ee.Princeton.edu

**Abstract.** The classical NP-complete problem of Boolean Satisfiability (SAT) has seen much interest in not just the theoretical computer science community, but also in areas where practical solutions to this problem enable significant practical applications. Since the first development of the basic search based algorithm proposed by Davis, Putnam, Logemann and Loveland (DPLL) about forty years ago, this area has seen active research effort with many interesting contributions that have culminated in state-of-the-art SAT solvers being able to handle problem instances with thousands and in some cases even millions of variables. In this paper we examine some of the main ideas along this passage that have led to our current capabilities. Given the depth of the literature in this field, it is impossible to do this in any comprehensive way; rather we focus on techniques with consistent demonstrated efficiency in available solvers. For the most part, we focus on techniques within the basic DPLL search framework, but also briefly describe other approaches and look at some possible future research directions.

## 1. Introduction

Given a propositional formula, determining whether there exists a variable assignment such that the formula evaluates to true is called the Boolean Satisfiability Problem, commonly abbreviated as SAT. SAT has seen much theoretical interest as the canonical NP-complete problem [1]. Given its NP-Completeness, it is very unlikely that there exists any polynomial algorithm for SAT. However, NP-Completeness does not exclude the possibility of finding algorithms that are efficient enough for solving many interesting SAT instances. These instances arise from many diverse areas - many practical problems in AI planning [2], circuit testing [3], software verification [4] can be formulated as SAT instances. This has motivated the research in practically efficient SAT solvers.

This research has resulted in the development of several SAT algorithms that have seen practical success. These algorithms are based on various principles such as resolution [5], search [6], local search and random walk [7], Binary Decision Diagrams [8], Stålmarck's algorithm [9], and others. Gu *et al.* [10] provide an excellent review of many of the algorithms developed thus far. Some of these algorithms are complete, while others are stochastic methods. For a given SAT instance, complete SAT solvers can either find a solution (i.e. a satisfying variable assignment) or prove that no solution exists. Stochastic methods, on the other hand, cannot prove the instance to be unsatisfiable even though they may be able to find a

solution for certain kinds of satisfiable instances quickly. Stochastic methods have applications in domains such as AI planning [2] and FPGA routing [11], where instances are likely to be satisfiable and proving unsatisfiability is not required. However, for many other domains (especially verification problems e.g. [4, 12]), the primary task is to prove unsatisfiability of the instances. For these, complete SAT solvers are a requirement.

In recent years search-based algorithms based on the well-known Davis-Logemann-Loveland algorithm [6] (sometimes called the DPLL algorithm for historical reasons) are emerging as some of the most efficient methods for complete SAT solvers. Researchers have been working on DPLL-based SAT solvers for about forty years. In the last ten years we have seen significant growth and success in SAT solver research based on the DPLL framework. Earlier SAT solvers based on DPLL include Tableau (NTAB) [13], POSIT [14], 2cl [15] and CSAT [16] among others. They are still appearing occasionally in the literature for performance comparison reasons. In the mid 1990's, Silva and Sakallah [17], and Bayardo and Schrag [18] proposed to augment the original DPLL algorithm with non-chronological backtracking and conflict-driven learning. These techniques greatly improved the efficiency of the DPLL algorithm for structured (in contrast to randomly generated) SAT instances. Many practical applications emerged (e.g. [4, 11, 12]), which pushed these solvers to their limits and provided strong motivation for finding even more efficient algorithms. This led to a new generation of solvers such as SATO [19], Chaff [20], and BerkMin [21] which pay a lot of attention to optimizing various aspects of the DPLL algorithm. The results are some very efficient SAT solvers that can often solve SAT instances generated from industrial applications with tens of thousands or even millions of variables. On another front, solvers such as satz [22] and cnfs [23] keep pushing the ability to tackle hard random 3-SAT instances. These solvers, though very efficient on random instances, are typically not competitive on structured instances generated from real applications.

A DPLL-based SAT solver is a relatively small piece of software. Many of the solvers mentioned above have only a few thousand lines of code (these solvers are mostly written in C or C++, for efficiency reasons). However, the algorithms involved are quite complex and a lot of attention is focused on various aspects of the solver such as coding, data structures, choosing algorithms and heuristics, and parameter tuning. Even though the overall framework is well understood and people have been working on it for years, it may appear that we have reached a plateau in terms of what can be achieved in practice – however we feel that many open questions still exist and present many research opportunities.

In this paper we chart the journey from the original basic DPLL framework through the introduction of efficient techniques within this framework culminating at current state-of-the-art solvers. Given the depth of literature in this field, it is impossible to do this in any comprehensive way; rather, we focus on techniques with consistent demonstrated efficiency in available solvers. While for the most part, we focus on techniques within the basic DPLL search framework, we will also briefly describe other approaches and look at some possible future research directions.

## 2. The Basic DPLL Framework

Even though there were many developments pre-dating them, the original algorithm for solving SAT is often attributed to Davis and Putnam for proposing a resolution-based algorithm for Boolean SAT in 1960 [5]. The original algorithm proposed suffers from the problem of memory explosion. Therefore, Davis, Logemann and Loveland [6] proposed a modified version that used search instead of resolution to limit the memory required for the solver. This algorithm is often referred to as the DPLL algorithm. It can be argued that intrinsically these two algorithms are tightly related because search (i.e. branching on variables) can be regarded as a special type of resolution. However, in the future discussion we will regard search-based algorithms as their own class and distinguish them from explicit resolution algorithms.

For the efficiency of the solver, the propositional formula instance is usually presented in a Product of Sum form, usually called a **Conjunctive Normal Form (CNF)**. It is not a limitation to require the instance to be presented in CNF. There exist polynomial algorithms (e.g. [24]) to transform any propositional formula into a CNF formula that has the same satisfiability as the original one. In the discussions that follow, we will assume that the problem is presented in CNF. A SAT instance in CNF is a logical **and** of one or more **clauses**, where each clause is a logical **or** of one or more **literals**. A literal is either the positive or the negative occurrence of a **variable**.

A propositional formula in CNF has some nice properties that can help prune the search space and speed up the search process. To satisfy a CNF formula, each clause must be satisfied individually. If there exists a clause in the formula that has all its literals assigned value 0, then the current variable assignment or any variable assignment that contains this will not be able to satisfy the formula. A clause that has all its literals assigned to value 0 is called a **conflicting clause**.

```
DPLL(formula, assignment) {
    necessary = deduction(formula, assignment);
    new_asgnmnt = union(necessary, assignment);
    if (is_satisfied(formula, new_asgnmnt))
        return SATISFIABLE;
    else if (is_conflicting(formula, new_asgnmnt))
        return CONFLICT;
    var = choose_free_variable(formula, new_asgnmnt);
    asgn1 = union(new_asgnmnt, assign(var, 1));
    if (DPLL(formula, asgn1) == SATISFIABLE)
        return SATISFIABLE;
    else
        asgn2 = union(new_asgnmnt, assign(var, 0));
        return DPLL(formula, asgn2);
}
```

Fig. 1. The recursive description of DPLL algorithm

Traditionally the DPLL algorithm is written in a recursive manner as shown in Fig. 1. Function `DPLL()` is called with a formula and a set of variable assignments. Function `deduction()` will return with a set of the necessary variable assignments that can be deduced from the existing variable assignments. The recursion will end if the formula is either satisfied (i.e. evaluates to 1 or **true**) or unsatisfied (i.e. evaluates to 0 or **false**) under the current variable assignment. Otherwise, the algorithm will choose an unassigned variable from the formula and branch on it for both phases. The solution process begins with calling the function `DPLL()` with an empty set of variable assignments.

In [25], the authors generalized many of the actual implementations of various solvers based on DPLL and rewrote it in an iterative manner as shown in Fig. 2. The algorithm described in Fig. 2 is an improvement of algorithm in Fig. 1 as it allows the solver to backtrack non-chronologically, as we will see in the following sections. Different solvers based on DPLL differ mainly in the detailed implementation of each of the functions shown in Fig. 2. We will use the framework of Fig. 2 as the foundation for our discussions that follow.

The algorithm described in Fig. 2 is a branch and search algorithm. Initially, none of the variables is assigned a value. We call unassigned variables **free** variables. First the solver will do some preprocessing on the instance to be solved, done by function `preprocess()` in Fig. 2. If preprocessing cannot determine the outcome, the main loop begins with a branch on a free variable by assigning it a value. We call this operation a **decision** on a variable, and the variable will have a **decision level** associated with it, starting from 1 and incremented with subsequent decisions. This is done by function `decide_next_branch()` in Fig. 2. After the branch, the problem is simplified as a result of this decision and its consequences. The function `deduce()` performs some reasoning to determine variable assignments that are needed for the problem to be satisfiable given the current set of decisions. Variables that are assigned as a consequence of this deduction after a branch will assume the same decision level as the decision variable. After the deduction, if all the clauses are satisfied, then the instance is satisfiable; if there exists a conflicting clause, then the

```

status = preprocess();
if (status != UNKNOWN) return status;
while(1) {
    decide_next_branch();
    while (true) {
        status = deduce();
        if (status == CONFLICT) {
            blevel = analyze_conflict();
            if (blevel == 0)
                return UNSATISFIABLE;
            else backtrack(blevel);
        }
        else if (status == SATISFIABLE)
            return SATISFIABLE;
        else break;
    }
}

```

*Handwritten annotations: A red circle highlights the conflict handling loop. A red checkmark is next to the `while (true)` loop. The word `UNKNOWN` is written in red at the end of the `else break;` line.*

**Fig. 2.** The iterative description of DPLL algorithm

current branch chosen cannot lead to a satisfying assignment, so the solver will backtrack (i.e. undo certain branches). Which decision level to backtrack to is determined by the function `analyze_conflict()`. Backtrack to level 0 indicates that even without any branching, the instance is still unsatisfiable. In that case, the solver will declare that the instance is unsatisfiable. Within the function `analyze_conflict()`, the solver may do some analysis and record some information from the current conflict in order to prune the search space for the future. This process is called **conflict-driven learning**. If the instance is neither satisfied nor conflicting under the current variable assignments, the solver will choose another variable to branch and repeat the process.

### 3. The Components of a DPLL SAT Solver

In this section of the paper, we discuss each of the components of a DPLL SAT solver. Each of these components has been the subject of much scrutiny over the years. This section focuses on the main lessons learnt in this process.

#### 3.1 The Branching Heuristics

Branching occurs in the function `decide_next_branch()` in Fig. 2. When no more deduction is possible, the function will choose one variable from all the free variables and assign it to a value. The importance of choosing good branching variables is well known - different branching heuristics may produce drastically different sized search trees for the same basic algorithm, thus significantly affect the efficiency of the solver. Over the years many different branching heuristics have been proposed by different researchers. Not surprisingly, comparative experimental evaluations have also been done (e.g. [26, 27]).

Early branching heuristics such as Bohm's Heuristic (reported in [28]), Maximum Occurrences on Minimum sized clauses (MOM) (e.g. [14]), and Jeroslow-Wang [29] can be regarded as greedy algorithms that try to make the next branch generate the largest number of implications or satisfy most clauses. All these heuristics use some functions to estimate the effect of branching on each free variable, and choose the variable that has the maximum function value. These heuristics work well for certain classes of instances. However, all of the functions are based on the statistics of the clause database such as clause length etc. These statistics, though useful for random SAT instances, usually do not capture relevant information about structured problems.

In [26], the author proposed the use of literal count heuristics. Literal count heuristics count the number of unresolved (i.e. unsatisfied) clauses in which a given variable appears in either phase. In particular, the author found that the heuristic that chooses the variable with dynamic largest combined sum (DLIS) of literal counts in both phases gives quite good results for the benchmarks tested. Notice that the counts are state-dependent in the sense that different variable assignments will give different counts. The reason is because whether a clause is unresolved (unsatisfied) depends on the current variable assignment. Because the count is state-dependent, each time the

function `decide_next_branch()` is called, the counts for all the free variables need to be recalculated.

As the solvers become more and more efficient, calculating counts for branching dominates the run time. Therefore, more efficient and effective branching heuristics are needed. In [20], the authors proposed the heuristic called Variable State Independent Decaying Sum (VSIDS). VSIDS keeps a score for each phase of a variable. Initially, the scores are the number of occurrences of a literal in the initial problem. Because modern SAT solvers have a learning mechanism, clauses are added to the clause database as the search progresses. VSIDS increases the score of a variable by a constant whenever an added clause contains the variable. Moreover, as the search progresses, periodically all the scores are divided by a constant number. In effect, the VSIDS score is a literal occurrence count with higher weight on the more recently added clauses. VSIDS will choose the free variable with the highest combined score to branch. Experiments show that VSIDS is quite competitive compared with other branching heuristics on the number of branches needed to solve a problem. Because VSIDS is state independent (i.e. scores are not dependent on the variable assignments), it is cheap to maintain. Experiments show that the decision procedure using VSIDS takes a very small percentage of the total run time even for problems with millions of variables.

More recently, [21] proposed another decision scheme that pushes the idea of VSIDS further. Like VSIDS, the decision strategy is trying to decide on the variables that are “active recently”. In VSIDS, the activity of a variable is captured by the score that is related to the literal’s occurrence. In [21], the authors propose to capture the activity by conflicts. More precisely, when a conflict occurs, all the literals in the clauses that are responsible for the conflict will have their score increased. A clause is responsible for a conflict if it is involved in the resolution process of generating the learned clauses (described in the following sections). In VSIDS, the focus on “recent” is captured by decaying the score periodically. In [21], the scores are also decayed periodically. Moreover, the decision heuristic will limit the decision variable to be among the literals that occur in the last added clause that is unresolved. The experiments seem to indicate that the new decision scheme is more robust compared with VSIDS on the benchmarks tested.

In other efforts, satz [22] proposed the use of look-ahead heuristics for branching; and cnfs [23] proposed the use of backbone-directed heuristics for branching. They share the common feature that they both seem to be quite effective on difficult random problems. However, they are also quite expensive compared with VSIDS. Random SAT problems are usually much harder than structured problems of the same size. Current solvers can only attack hard random 3-SAT problems with several hundred variables. Therefore, the instances regarded as hard for random SAT is generally much smaller in size than the instances considered hard for structured problems. Thus, while it may be practical to apply these expensive heuristics to the smaller random problems, their overhead tends to be unacceptable for the larger well-structured problems.

### 3.2 The Deduction algorithm

Function `deduce()` serves the purpose of pruning the search space by “look ahead”. When a branch variable is assigned a value, the entire clause database is simplified. Function `deduce()` needs to determine the consequences of the last decision to make the instance satisfiable, and may return three status values. If the instance is satisfied under the current variable assignment, it will return `SATISFIABLE`; if the instance contains a conflicting clause, it will return `CONFLICT`; otherwise, it will return `UNKNOWN` and the solver will continue to branch. There are various mechanisms with different deduction power and run time costs for the deduce function. The correctness of the algorithm will not be affected as long as the deduction rules incorporated are valid (e.g. it will not return `SATISFIABLE` when the instance contains a conflicting clause under the assignment). However, different deduction rules, or even different implementations of the same rule, can significantly affect the efficiency of the solver.

Over the years several different deduction mechanisms have been proposed. However, it seems that the unit clause rule [6] is the most efficient one because it requires relatively little computational power but can prune large search spaces. The unit clause rule states that for a certain clause, if all but one of its literals has been assigned the value 0, then the remaining (unassigned) literal must be assigned the value 1 for this clause to be satisfied, which is essential for the formula to be satisfied. Such clauses are called **unit clauses**, and the unassigned literal in a unit clause is called a **unit literal**. The process of assigning the value 1 to all unit literals is called **unit propagation**, or sometimes called **Boolean Constraint Propagation (BCP)**. Almost all modern SAT solvers incorporate this rule in the deduction process. In a SAT solver, BCP usually takes the most significant part of the run time. Therefore, its efficiency is directly related to the implementation of the BCP engine.

#### 3.2.1 BCP Mechanisms

In a SAT solver, the BCP engine's function is to detect unit clauses and conflicting clauses after a variable assignment. The BCP engine is the most important part of a SAT solver and usually dictates the data structure and organization of the solver.

A simple and intuitive implementation for BCP is to keep counters for each clause. This scheme is attributed to Crawford and Auton [13] by [30]. Similar schemes are subsequently employed in many solvers such as GRASP [25], `rel_sat` [18], `satz` [22] etc. For example, in GRASP [25], each clause keeps two counters, one for the number of value 1 literals in the clause and the other for the number of value 0 literals in the clause. Each variable has two lists that contain all the clauses where that variable appears as a positive and negative literal, respectively. When a variable is assigned a value, all the clauses that contain this literal will have their counters updated. If a clause's value 0 count becomes equal to the total number of literals in the clause, then it is a conflicting clause. If a clause's value 0 count is one less than the total number of literals in the clause and the value 1 count is 0, then the clause is a unit clause. A counter-based BCP engine is easy to understand and implement, but this scheme is not the most efficient one. If the instance has  $m$  clauses and  $n$  variables, and on average each clause has  $l$  literals, then whenever a variable gets assigned, on the

average  $l m / n$  counters need to be updated. On backtracking from a conflict, we need to undo the counter assignments for the variables unassigned during the backtracking. Each undo for a variable assignment will also update  $l m / n$  counters on average. Modern solvers usually incorporate learning mechanisms in the search process (described in the next sections), and learned clauses often have many literals. Therefore, the average clause length  $l$  is quite large, thus making a counter-based BCP engine relatively slow.

In [30], the authors of the solver SATO proposed the use of another mechanism for BCP using head/tail lists. In this mechanism, each clause has two pointers associated with it, called the head and tail pointer respectively. A clause stores all its literals in an array. Initially, the head pointer points to the first literal of the clause (i.e. beginning of the array), and the tail pointer points to the last literal of the clause (i.e. end of the array). Each variable keeps four linked lists that contain pointer to clauses. The linked lists for the variable  $v$  are `clause_of_pos_head(v)`, `clause_of_neg_head(v)`, `clause_of_pos_tail(v)` and `clause_of_neg_tail(v)`. Each of these lists contains the pointers to the clauses that have their head/tail literal in positive/negative phases of variable  $v$ . If  $v$  is assigned with the value 1, `clause_of_pos_head(v)` and `clause_of_pos_tail(v)` will be ignored. For each clause  $C$  in `clause_of_neg_head(v)`, the solver will search for a literal that does not evaluate to 1 from the position of the head literal of  $C$  to the position of the tail literal of  $C$ . Notice the head literal of  $C$  must be a literal corresponding to  $v$  in negative phase. During the search process, four cases may occur:

- 1) If during the search we first encounter a literal that evaluates to 1, then the clause is satisfied, we need to do nothing.
- 2) If during the search we first encounter a literal  $l$  that is free and  $l$  is not the tail literal, then we remove  $C$  from `clause_of_neg_head(v)` and add  $C$  to head list of the variable corresponding to  $l$ . We refer to this operation as moving the head literal, because in essence the head pointer is moved from its original position to the position of  $l$ .
- 3) If all literals in between these two pointers are assigned value 0, but the tail literal is unassigned, then the clause is a unit clause, and the tail literal is the unit literal for this clause.
- 4) If all literals in between these two pointers and the tail literal are assigned value 0, then the clause is a conflicting clause.

Similar actions are performed for `clause_of_neg_tail(v)`, only the search is in the reverse direction (i.e. from tail to head).

Head/tail list method is faster than the counter-based scheme because when the variable is assigned value 1, the clauses that contain the positive literals of this clause will not be visited at all and vice-versa. As each clause has only two pointers, whenever a variable is assigned a value, the status of only  $m/n$  clauses needs to be updated on the average, if we assume head/tail literals are distributed evenly in either phase. Even though the work needed to be done for each update is different from the counter-based mechanism, in general head/tail mechanism is still much faster.

For both the counter-based algorithm and the head/tail list-based algorithm, undoing a variable's assignment during backtrack has about the same computational complexity as assigning the variable. In [20], the authors of the solver Chaff proposed



another BCP algorithm called 2-literal watching. Similar to the head/tail list algorithm, 2-literal watching also has two special literals for each clause called *watched literals*. Each variable has two lists containing pointers to all the watched literals corresponding to it in either phase. We denote the lists for variable  $v$  as  $\text{pos\_watched}(v)$  and  $\text{neg\_watched}(v)$ . In contrast to the head/tail list scheme in SATO, there is no imposed order on the two pointers within a clause, and each of the pointers can move in either direction. Initially the watched literals are free. When a variable  $v$  is assigned value 1, for each literal  $p$  pointed to by a pointer in the list of  $\text{neg\_watched}(v)$  (notice  $p$  must be a literal of  $v$  with negative phase), the solver will search for a literal  $l$  in the clause containing  $p$  that is not set to 0. There are four cases that may occur during the search:

- 1) If there exists such a literal  $l$  and it is not the other watched literal, then we remove pointer to  $p$  from  $\text{neg\_watched}(v)$ , and add pointer to  $l$  to the watched list of the variable corresponding to  $l$ . We refer to this operation as moving the watched literal, because in essence one of the watched pointers is moved from its original position to the position of  $l$ .
- 2) If the only such  $l$  is the other watched literal and it is free, then the clause is a unit clause, with the other watched literal being the unit literal.
- 3) If the only such  $l$  is the other watched literal and it evaluates to 1, then we need to do nothing.
- 4) If all literals in the clause is assigned value 0 and no such  $l$  exists, then the clause is a conflicting clause.

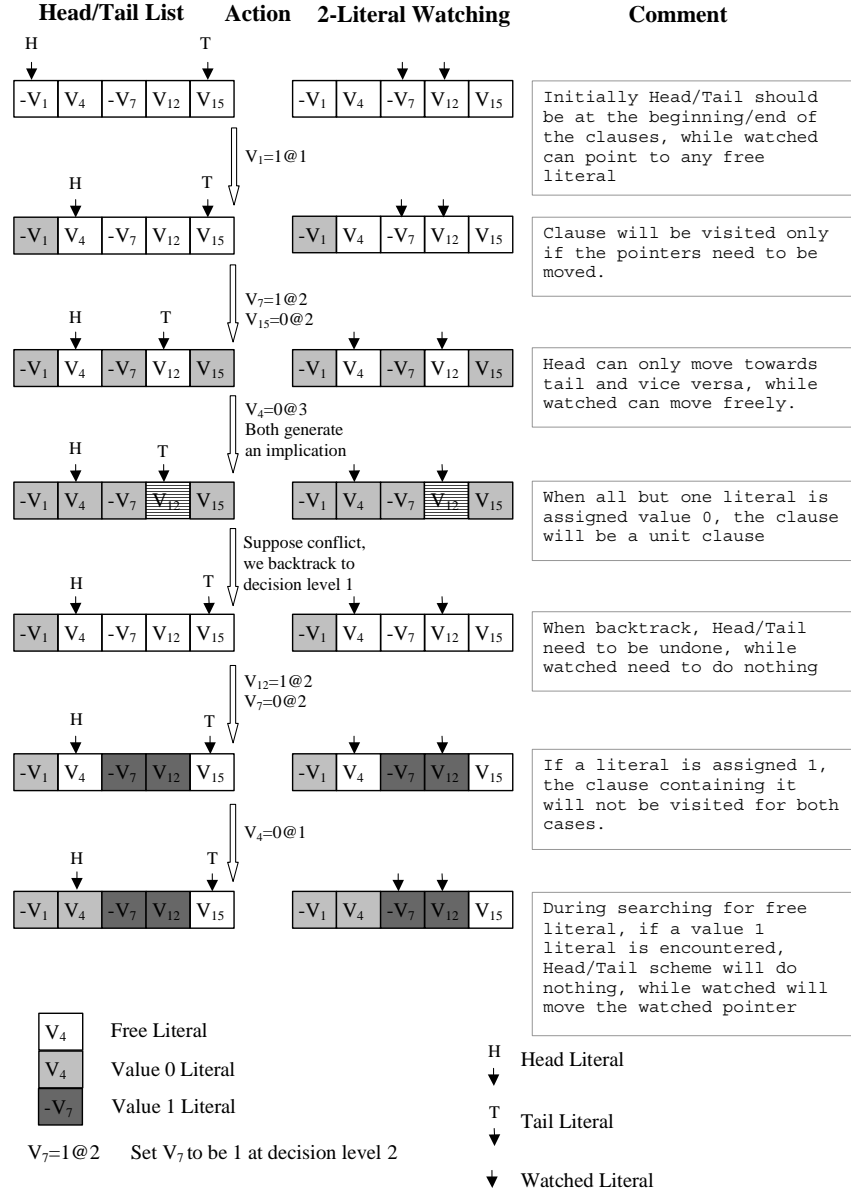
2-literal watching has the same advantage as the head/tail list mechanism compared with the literal counting scheme. Moreover, unlike the other two mechanisms, undoing a variable assignment during backtrack in the 2-literal watching scheme takes constant time. This is because the two watched literals are the last to be assigned to 0, so as a result, any backtracking will make sure that the literals being watched are either unassigned, or assigned to one. Thus, no action is required to update the pointers for the literals being watched. Therefore, it is significantly faster than both counter-based and head/tail mechanisms for BCP. In Fig. 3, we show a comparison of 2-literal watching and head/tail list mechanism.

In [31], the authors examined the mechanisms mentioned above and introduced some new deduction data structures and mechanisms. In particular, the experiments suggest that the mechanism called Head/Tail list with Literal Sifting actually outperforms the 2-literal watching mechanism for BCP. However, the experiments are carried out in a framework implemented in Java. The authors admit that it may not represent the actual performance if implemented in C/C++.

### 3.2.2 Other Deduction Mechanisms

Besides the unit clause rule, there are other rules that can be incorporated into a deduction engine. In this section, we briefly discuss some of them. We want to point out that though many of the deduction mechanisms have been shown to work on certain classes of SAT instances, unlike the unit clause rule, none of them seems to work without deteriorating the overall performance of the SAT solver for general SAT instances.

One of the most widely known rules for deduction is the **pure literal rule** [6]. The pure literal rule states that if a variable only occurs in a single phase in all the unresolved clauses, then it can be assigned with a value such that the literal of the variable in that phase evaluates to 1. Whether a variable satisfies the pure literal rule



**Fig. 3.** Comparison of Head/Tail List and 2-Literal Watching

is expensive to detect during the actual solving process, and the consensus seems to be that incorporating the pure literal rule will generally slow down the solving process for most of the benchmarks encountered.

Another explored deduction mechanism is equivalence reasoning. In particular, eqsatz [32] incorporated equivalence reasoning into the satz [22] solver and found that it is effective on some particular classes of benchmarks. In that work, the equivalence reasoning is accomplished by a pattern-matching scheme for equivalence clauses. A related deduction mechanism was proposed in [33]. There, the authors propose to include more patterns in the matching process for simplification purpose in deduction.

The unit literal rule basically guarantees that all the unit clauses are consistent with each other. We can also require that all the 2 literal clauses be consistent with each other and so on. Researchers have been exploring this idea in the deduction process in works such as [34, 35]. In particular, these approaches maintain a transitive closure of all the 2 literal clauses. However, the overhead of maintaining this information seems to far outweigh any benefit gained from them on the average.

Recursive Learning [36] is another reasoning technique originally proposed in the context of learning with a logic circuit representation of a formula. Subsequent research [37] has proposed to incorporate this technique in SAT solvers and found that it works quite well for some benchmarks generated from combinational circuit equivalence checking problems.

### 3.3 Conflict Analysis and Learning

When a conflicting clause is encountered, the solver needs to backtrack and undo the decisions. Conflict analysis is the procedure that finds the reason for a conflict and tries to resolve it. It tells the SAT solver that there exists no solution for the problem in a certain search space, and indicates a new search space to continue the search.

The original DPLL algorithm proposed the simplest conflict analysis method. For each decision variable, the solver keeps a flag indicating whether it has been tried in both phases (i.e. flipped) or not. When a conflict occurs, the conflict analysis procedure looks for the decision variable with the highest decision level that has not been flipped, marks it flipped, undoes all the assignments between that decision level and current decision level, and then tries the other phase for the decision variable. This method is called **chronological backtracking** because it always tries to undo the last decision that is not flipped. Chronological backtracking works well for random generate SAT instances and is employed in some SAT solvers (e.g satz [22]).

For structured problems (which is usually the case for problems generated from real world applications), chronological backtracking is generally not efficient in pruning the search space. More advanced conflict analysis engines will analyze the conflicting clauses encountered and figure out the direct reason for the conflict. This method will usually backtrack to an earlier decision level than the last unflipped decision. Therefore, it is called **non-chronological backtracking**. During the conflict analysis process, information about the current conflict may be recorded as clauses and added to the original database. The added clauses, though redundant in the sense that they will not change the satisfiability of the original problem, can often help to

prune search space in the future. This mechanism is called **conflict-directed learning**. Such learned clauses are called **conflict clauses** as opposed to **conflicting clauses**, which refer to clauses that generate conflicts.

Non-chronological backtracking, sometimes referred to as **conflict-directed backjumping**, was proposed first in the Constraint Satisfaction Problem (CSP) domain (e.g. [38]). This, together with conflict-directed learning, were first incorporated into a SAT solver by Silva and Sakallah in GRASP [25], and by Bayardo and Schrag in *rel\_sat* [18]. These techniques are essential for efficient solving of structured problems. Many solvers such as SATO [19] and Chaff [20] have incorporated similar technique in the solving process.

Previously, learning and non-chronological backtracking have been discussed by analyzing implication graphs (e.g. [17, 39]). Here we will formulate learning as an alternate but equivalent resolution process and discuss different schemes in this framework.

Researchers have adapted the conflict analysis engine to some deduction rules other than the unit clause rule in previous work (e.g. [33, 37]). However, because the unit clause rule is usually the only rule that is incorporated in most SAT solvers, we will describe the learning algorithm that works with such a deduction engine. In such a solver, when a variable is implied by a **unit clause**, the clause is called the **antecedent** of the variable. Because the unit clause rule is the only rule in the deduction engine, every implied variable will have an antecedent. Decision variables, on the other case, have no antecedents.

In conflict driven learning, the learned clauses are generated by resolution. Resolution is a process to generate a clause from two clauses analogous to the process of **consensus** in the logic optimization domain (e.g. [40]). Resolution is given by

$$(x + y) (y' + z) \equiv (x + y) (y' + z)(x + z)$$

The term  $(x + z)$  is called the **resolvent** of clause  $(x + y)$  and  $(y' + z)$ . Because of this, we have

$$(x + y) (y' + z) \rightarrow (x + z)$$

Similar to the well-known consensus law (e.g. [40]), the resulting clause of resolution between two clauses is redundant with respect to the original clauses. Therefore, we can always generate clauses from original clause database by resolution and add the generated clause back to the clause database without changing the satisfiability of the original formula. However, randomly choosing two clauses and adding the resolvent to the clause database will not generally help the solving process. Conflict-driven learning is a way to generate learned clauses with some direction in the resolution process.

The pseudo-code for conflict analysis is shown in Fig. 4. Whenever a conflicting clause is encountered, `analyze_conflict()` will be called. Function `choose_literal()` will choose a literal from the clause. Function `resolve(c11, c12, var)` will return a clause that contains all the literals in `c11` and `c12` except for the literals that corresponds to variable `var`. Note that one of the input clauses to `resolve()` is a conflicting clause (i.e. all literals evaluate to 0), and the other is the **antecedent of the variable `var`** (i.e. all but one literal evaluate to 0). Therefore, the resulting clause will have all literals evaluating to 0, i.e. it will still be a **conflicting clause**.

```

analyze_conflict(){
    cl = find_conflicting_clause();
    while (!stop_criterion_met(cl)) {
        lit = choose_literal(cl);
        var = variable_of_literal( lit );
        ante = antecedent( var );
        cl = resolve(cl, ante, var);
    }
    add_clause_to_database(cl);
    back_dl = clause_asserting_level(cl);
    return back_dl;
}

```

**Fig. 4.** Generating Learned Clause by Resolution

The clause generation process will stop when some predefined stop criterion is met. In modern SAT solvers, the stop criterion is that the resulting clause be an **asserting clause**. A clause is asserting if the clause contains all value 0 literals; and among them **only one is assigned at current decision level**. After backtracking, this clause will become a unit clause and force the literal to assume another value (i.e. evaluate to 1), thus bringing the search to a new space. We will call the decision level of the literal with the **second highest decision level** in an asserting clause the **asserting level of** that clause. **The asserting clause is a unit clause at its asserting decision level.**

In addition to the above asserting clause requirement, different learning schemes may have some additional requirements. Different learning schemes differ in their stop criterion and the way to choose literals. Notice the stop criterion can always be met if function `choose_literal()` **always chooses the literal that is assigned last in the clause**. If that is the case, the resolution process will always resolve the conflicting clause with the antecedent of the variable that is assigned last in the clause. After a certain number of calls to `resolve()`, there will always be a time when the variable that is assigned last in the clause is the decision variable of the current decision level. At this time, the resulting clause is guaranteed to be an asserting clause. The SAT solver `rel_sat` [18] actually uses this stop criterion, i.e. it requires that the variable that has the highest decision level in the resulting clause be a decision variable. The literal corresponding to this variable will be a unit literal after backtracking, resulting in essentially flipping the decision variable.

In [39], the authors discussed a scheme called the FirstUIP scheme. The FirstUIP scheme is quite similar to the `rel_sat` scheme but the stop criterion is that it will stop when the *first* asserting clause is encountered. In [17], the authors of GRASP use a similar scheme as the FirstUIP, but add extra clauses other than the asserting clause into the database. If function `choose_literal()` does not choose literals in reversed chronological order, then extra mechanisms are needed to guarantee that the stop criterion can be met. Some of the schemes discussed in [39] may need function `choose_literal()` to choose literals that are not in the current decision level.

Different learning schemes affect the SAT solver's efficiency greatly. Experiments in [39] show that among all the discussed schemes, FirstUIP seems to be the best on the benchmarks tested. Therefore, recent SAT solvers (e.g. Chaff [20]) often employ this scheme as the default conflict-driven learning scheme.

Conflict-driven learning will add clauses to the clause database during the search process. Because added clauses are redundant, deleting some or all of them will not affect the correctness of the algorithm. In fact, the added clauses will slow down the deduction engine, and keeping all added clauses may need more memory for storage than the available memory. Therefore, it is often required for the solver to delete some of the less useful learned clauses and learned clauses that have too many literals. There are many heuristics to measure the usefulness of a learned clause. For example, `rel_sat` [18] proposes to use relevance to measure a clause's usefulness, while `BerkMin` [21] use the number of conflicts that involve this clause in the history to measure a clause's usefulness. These measures seem to work reasonably well.

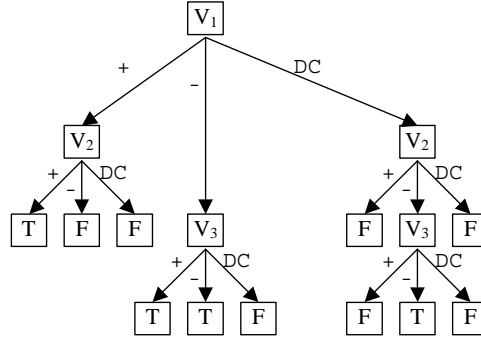
### 3.4 Data Structure for Storing Clause Database

Current state-of-the-art SAT solvers often need to solve instances that are quite large in size. Some instances generated from circuit verification problems may contain millions of variables and several million clauses. Moreover, during the SAT solving process, learned clauses are generated for each conflict encountered and may further increase the dataset size. Therefore, efficient data structures for storing the clauses are needed.

Most commonly, clauses are stored in a linear way (sometimes called **sparse matrix representation**), i.e. each clause occupies its own space and no overlap exists between clauses. Therefore, the dataset size is linear in the number of literals in the clause database. Early SAT solvers (e.g. `GRASP` [25], `rel_sat` [18]) use **pointer heavy** data structures such as linked lists and array of pointers pointing to structures to store the clause database. Pointer heavy data structures, though convenient for manipulating the clause database (i.e. adding/deleting clauses), are not memory efficient and usually cause a lot of **cache misses** during the solving process because of lack of access locality. `Chaff` [20] uses a data structure that stores clause data in a **large array**. Because arrays are not as flexible as linked lists, some additional **garbage collection** code is needed when clauses are deleted. The advantage of the array data structure is that it is very efficient in memory utilization. Moreover, because an array occupies contiguous memory space, access locality is increased. Experiments shows that the array data structure has a big advantage compared with linked lists in terms of **cache misses** that translates to substantial speed-up in the solving process.

Researchers have proposed schemes other than sparse matrix representation for storing clauses. In [41], the authors of the solver `SATO` proposed the use of a data structure called **trie** to store clauses. A trie is a ternary tree. Each internal node in the trie structure is a variable index, and its three children edges are labeled Pos, Neg, and DC, for positive, negative, and don't care, respectively. A leaf node in a trie is either True or False. Each path from root of the trie to a True leaf represents a clause. A trie is said to be ordered if for every internal node  $V$ ,  $\text{Parent}(V)$  has a smaller variable index than the index of variable  $V$ . The ordered trie structure has the nice property of being able to detect duplicate and tail subsumed clauses of a database quickly. A clause is said to be tail subsumed by another clause if its first portion of the literals (a prefix) is also a clause in the clause database. For example,  $(a + b + c)$  is tail

subsumed by  $(a + b)$ . Fig. 5 shows a simple clause database represented in a trie structure.



**Fig. 5.** A trie data structure representing clauses  $(V_1+V_2)$   $(V_1'+V_3)$   $(V_1'+V_3')$   $(V_2'+V_3')$

An ordered trie has obvious similarities with Binary Decision Diagrams. This has naturally led to the exploration of decision diagram style set representations. In [42] and [43], the authors have experimented with using Zero-suppressed Binary Decision Diagrams (ZBDDs) [44] to represent the clause database. A ZBDD representation of the clause database can detect not only tail subsumption but also head subsumption. Both authors report significant compression of the clause database for certain classes of problems.

Based on current experimental data it does not seem that the data compression advantages of the trie and ZBDD data structures are sufficient to justify the additional maintenance overhead of these data structures compared to the sparse matrix representation.

### 3.5 Preprocess, Restart and other techniques

Preprocess aims at simplifying the instances before the regular solving begins in order to speed up the solving process. Usually the preprocessor of a SAT solver is just an extra deduction mechanism applied at the beginning of the search. Because the preprocessor will only be applied once in the solving process, it is usually possible to incorporate some deduction rules that are too expensive to be applied at every node of the search tree. The preprocessor can be applied on-line (within the solver) or off-line (it produces an equivalent instance to be fed to a solver). In [45], the authors give an overview of some of the existing preprocessing techniques and find that the result of applying simplification techniques before the regular search is actually mixed.

The time required for solving similar SAT instances often varies greatly for complete algorithms. Two problems that are exactly the same except for the variable order may require totally different times to solve by a certain SAT solver (e.g. one can be solved in seconds while the other takes days). In [46], the authors proposed to use **random restart** to cope with this phenomenon. Random restart randomly throws away the already searched space and starts from scratch. This technique is applied in

modern SAT solvers such as Chaff [20] and BerkMin [21]. In these cases, when restart is invoked, even though the current search tree is abandoned, because the solver still keeps some of the learned clauses, the previous search effort is not totally lost. Experiments show that random restarts can increase the robustness of certain SAT solvers.

Researchers have been extending the randomization idea of random restart to other aspects of the SAT solving process as well. For example, portfolio design [47] aims at using different solving strategies during one solving process in order to make the solver robust. Some researchers [48] also propose to randomize backtracking. All in all, it seems that randomization is quite important because of the heavy tail [49] nature of SAT solving process.

#### **4. Other techniques used in SAT solvers**

In this section, we briefly discuss some of the other techniques used to solve SAT problems besides the basic DPLL search.

The original Davis Putnam algorithm [5] was based on resolution. A well-known problem of the resolution-based algorithm is that the solver tends to blow up in memory. Because of this, resolution based algorithm is seldom employed in modern SAT solvers. In [42], the authors propose the use of ZBDDs to represent clauses in a resolution-based solver and utilize the compression power of decision diagrams to control the memory blowup. Their experiment shows that for certain classes of SAT problems, the resolution-based approach shows very good results.

Stålmark's algorithm [9] is a patented proprietary algorithm for solving SAT. Stålmark's algorithm use breath-first search in contrast to the depth-first search employed by DPLL. There are commercial implementations of SAT solvers based on this algorithm [50]. HeerHugo [51] is a publicly available solver that claims to be using an algorithm similar to the Stålmark's algorithm.

Another approach is to use stochastic algorithms. Stochastic algorithms cannot prove a SAT instance to be unsatisfiable. However, for some hard satisfiable instances, stochastic methods may find solutions very quickly. Currently, two of the more successful approaches to the stochastic method are random walk based algorithms such as walksat [7] and Discrete Lagrangian-Based global search methods such as DLM [52].

For more about other SAT solving techniques, we refer the readers to a survey[10].

#### **5. Conclusions and Future Works**

In this paper, we briefly discussed some of the techniques employed in modern Boolean Satisfiability solvers. In particular, we concentrated on the procedure based on the DPLL search algorithm. In recent years, SAT solvers based on DPLL search have made phenomenal progress. Efficient SAT solvers such as Chaff [20] are deployed in industrial strength applications for hardware verification and debugging. In these environments, the SAT solver routinely encounters instances with thousands



or even millions of variables. Therefore, it is of great importance to increase the capacity and efficiency of the SAT solver.

Even though researchers have been working on SAT engines for quite a long time, there is still a lot of work that remains to be done. First of all, the overall understanding of SAT instances is still quite limited. For example, though there exist some rough ideas about the difficulty of SAT problems (e.g. [53, 54]), it is still not clear how can we estimate the hardness of a given problem without actually solving it. Experimental evaluation of different SAT solving algorithms is more like an art than a science because it is easy to tune a solver to a given set of benchmarks, but the parameters may not work for the same benchmarks with some simple permutation (e.g.[55]). On the application side, currently most of the applications use SAT solvers as blackboxes and no interaction is possible between the applications and the SAT solvers. Application specific knowledge can help a lot in the solving process as demonstrated in [56]. For a particular application, custom implementation of a SAT solver may also be helpful (e.g. [57]). All in all, we believe there are still many research topics to be explored. As more and more applications utilize SAT solvers as deduction and reasoning engine, we believe many new algorithms will emerge and push the envelope for efficient implementations even further.

## Acknowledgments

The authors would like to thank Dr. Aarti Gupta for suggestions and help in improving the paper.

## 6. References

- [1] S. A. Cook, "The complexity of theorem-proving procedures," presented at Third Annual ACM Symposium on Theory of Computing, 1971.
- [2] H. Kautz and B. Selman, "Planning as Satisfiability," presented at European Conference on Artificial Intelligence(ECAI-92), 1992.
- [3] P. Stephan, R. Brayton, and A. Sangiovanni-Vencentelli, "Combinational Test Generation Using Satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 1167-1176, 1996.
- [4] D. Jackson and M. Vaziri, "Finding Bugs with a Constraint Solver," presented at International Symposium on Software Testing and Analysis, Portland, OR, 2000.
- [5] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of ACM*, vol. 7, pp. 201-215, 1960.
- ~~[6]~~ M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394-397, 1962.
- [7] B. Selman, H. Kautz, and B. Cohen, "Local Search Strategies for Satisfiability Testing," in *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 26, D. S. Johnson and M. A. Trick, Eds.: American Mathematical Society, 1996.
- [8] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677-691, 1986.

- [9] G. Stålmarck, "A system for determining prepositional logic theorems by applying values and rules to triplets that are generated from a formula." US Patent N 5 27689, 1995.
- [10] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*: American Mathematical Society, 1997.
- [11] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar, "Satisfiability-Based Layout Revisited: Detailed Routing of Complex FPGAs Via Search-Based Boolean SAT," presented at ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'99), Monterey, California, 1999.
- [12] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," presented at Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), 1999.
- [13] J. Crawford and L. Auton, "Experimental results on the cross-over point in satisfiability problems," presented at National Conference on Artificial Intelligence (AAAI), 1993.
- [14] J. W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," in *Ph.D. Thesis, Department of Computer and Information Science*: University of Pennsylvania, 1995.
- [15] A. V. Gelder and Y. K. Tsuji, "Satisfiability Testing with more Reasoning and Less guessing," in *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, M. Trick, Ed.: American Mathematical Society, 1995.
- [16] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier, "SAT v.s. UNSAT," in *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, D. S. Johnson and M. Trick, Eds., 1993.
- [17] J. P. Marques-Silva and K. A. Sakallah, "Conflict Analysis in Search Algorithms for Propositional Satisfiability," presented at IEEE International Conference on Tools with Artificial Intelligence, 1996.
- [18] R. Bayardo and R. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," presented at National Conference on Artificial Intelligence (AAAI), 1997.
- [19] H. Zhang, "SATO: An efficient propositional prover," presented at International Conference on Automated Deduction (CADE), 1997.
- [20] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," presented at 39th Design Automation Conference, 2001.
- [21] E. Goldberg and Y. Novikov, "BerkMin: a Fast and Robust SAT-Solver," presented at Design Automation & Test in Europe (DATE 2002), 2002.
- [22] C. M. Li and Anbulagan, "Heuristics based on unit propagation for satisfiability problems," presented at the fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagayo, Japan, 1997.
- [23] O. Dubois and G. Dequen, "A backbone-search heuristic for efficient solving of hard 3-SAT formulae," presented at International Joint Conference on Artificial Intelligence (IJCAI), 2001.
- [24] D. A. Plaisted and S. Greenbaum, "A Stucture-preserving Clause Form Translation," *Journal of Symbolic Computation*, vol. 2, pp. 293-304, 1986.
- [25] J. P. Marques-Silva and K. A. Sakallah, "GRASP -- A New Search Algorithm for Satisfiability," presented at IEEE International Conference on Tools with Artificial Intelligence, 1996.
- [26] J. P. Marques-Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," presented at the 9th Portuguese Conference on Artificial Intelligence (EPIA), 1999.
- [27] J. N. Hooker and V. Vinay, "Branching rules for satisfiability," *Journal of Automated Reasoning*, vol. 15, pp. 359-383, 1995.

- [28] M. Buro and H. Kleine-Buning, "Report on a SAT competition," Technical Report, University of Paderborn 1992.
- [29] R. G. Jeroslow and J. Wang, "Solving propositional satisfiability problems," *Annals of Mathematics and Artificial Intelligence*, vol. 1, pp. 167-187, 1990.
- [30] H. Zhang and M. Stickel, "An efficient algorithm for unit-propagation," presented at International Symposium on Artificial Intelligence and Mathematics, Ft. Lauderdale, Florida, 1996.
- [31] I. Lynce and J. P. Marques-Silva, "Efficient data structures for backtrack search SAT solvers," presented at Fifth International Symposium on the Theory and Applications of Satisfiability Testing, 2002.
- [32] C. M. Li, "Integrating equivalency reasoning into Davis-Putnam Procedure," presented at National Conference on Artificial Intelligence (AAAI), 2000.
- [33] I. Lynce and J. P. Marques-Silva, "Integrating Simplification Techniques in SAT Algorithms," presented at Logic in Computer Science Short Paper Session (LICS-SP), 2001.
- [34] A. V. Gelder and Y. K. Tsuji, "Satisfiability Testing with more Reasoning and Less guessing," in *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, D. S. Johnson and M. Trick, Eds.: American Mathematical Society, 1993.
- [35] S. T. Chakradhar and V. D. Agrawal, "A Transitive Closure Based Algorithm for Test Generation," presented at Design Automation Conference (DAC), 1991.
- [36] W. Kunz and D. K. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 1143-1158, 1994.
- [37] J. P. Marques-Silva, "Improving Satisfiability Algorithms by Using Recursive Learning," presented at International Workshop on Boolean Problems (IWBP), 1998.
- [38] P. Prosser, "Hybrid algorithms for the constraint satisfaction problem," *Computational Intelligence*, vol. 9, pp. 268-299, 1993.
- [39] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," presented at International Conference on Computer Aided Design (ICCAD), San Jose, CA, 2001.
- [40] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*: Kluwer Academic Publishers, 1996.
- [41] H. Zhang and M. Stickel, "Implementing Davis-Putnam's method," Technical Report, University of Iowa 1994.
- [42] P. Chatalic and L. Simon, "Multi-Resolution on Compressed Sets of Clauses," presented at International Conference on Tools with Artificial Intelligence, 2000.
- [43] F. Aloul, M. Mneimneh, and K. Sakallah, "Backtrack Search Using ZBDDs," presented at International Workshop on Logic Synthesis (IWLS), 2001.
- [44] S. I. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," presented at 30th Design Automation Conference (DAC), 1993.
- [45] I. Lynce and J. P. Marques-Silva, "The Puzzling Role of Simplification in Propositional Satisfiability," presented at EPIA'01 Workshop on Constraint Satisfaction and Operational Research Techniques for Problem Solving (EPIA-CSOR), 2001.
- [46] C. P. Gomes, B. Selman, and H. Kautz, "Boosting Combinatorial Search Through Randomization," presented at National Conference on Artificial Intelligence (AAAI), Madison, WI, 1998.
- [47] B. A. Huberman, R. M. Lukose, and T. Hogg, "An Economics approach to hard computational problems," *Science*, vol. 275, pp. 51-54, 1997.
- [48] I. Lynce and J. P. Marques-Silva, "Complete unrestricted backtracking algorithms for Satisfiability," presented at Fifth International Symposium on the Theory and Applications of Satisfiability Testing, 2002.

- [49] C. P. Gomes, B. Selman, N. Crator, and H. Kautz, "Heavy-tailed phenomena in satisfiability and constraint satisfaction problems," *Journal of Automated Reasoning*, vol. 24(1/2), pp. 67-100, 1999.
- [50] "Prover Proof Engine," Prover Technology.
- [51] J. F. Groote and J. P. Warners, "The propositional formula checker HeerHugo," *Journal of Automated Reasoning*, vol. 24, 2000.
- [52] Y. Shang and B. W. Wah, "A Discrete Lagrangian-Based Global-Search Method for Solving Satisfiability Problems," *Journal of Global Optimization*, vol. 12, pp. 61-99, 1998.
- [53] I. Gent and T. Walsh, "The SAT Phase Transition," presented at European Conference on Artificial Intelligence (ECAI-94), 1994.
- [54] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?," presented at Design Automation Conference (DAC99), 1999.
- [55] F. Brglez, X. Li, and M. Stallmann, "The role of a skeptic agent in testing and benchmarking of SAT algorithms," presented at Fifth International Symposium on the Theory and Applications of Satisfiability Testing, 2002.
- [56] O. Strichman, "Pruning techniques for the SAT-based Bounded Model Checking Problem," presented at 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARM'01), 2001.
- [57] M. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik, "Combining Strengths of Circuit-based and CNF-based Algorithms for a High-Performance SAT Solver," presented at Design Automation Conference (DAC'02), 2002.