Chapter 9
Propositional SAT Solving

Joao Marques-Silva and Sharad Malik

Abstract The Boolean Satisfiability Problem (SAT) is well known in computational complexity, representing the first decision problem to be proved NP-complete. SAT is also often the subject of work in proof complexity. Besides its theoretical interest, SAT finds a wide range of practical applications. Moreover, SAT solvers have been the subject of remarkable efficiency improvements since the mid-1990s, motivating their widespread use in many practical applications including Bounded and Unbounded Model Checking. The success of SAT solvers has also motivated the development of algorithms for natural extensions of SAT, including Quantified Boolean Formulas (QBF), Pseudo-Boolean constraints (PB), Maximum Satisfiability (MaxSAT) and Satisfiability Modulo Theories (SMT) which also see application in the model-checking context. This chapter first covers the organization of modern conflict-driven clause learning (CDCL) SAT solvers, which are used in the vast majority of practical applications of SAT. It then reviews the techniques shown to be effective in modern SAT solvers.

9.1 Introduction

Given a propositional logic formula, determining whether there exists a variable assignment such that the formula evaluates to true is referred to as the Boolean Satisfiability Problem, commonly abbreviated as SAT. SAT has seen much theoretical interest as the canonical NP-complete problem [30]. 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 to solve many interesting SAT instances. In addition to model checking, the subject of this book, these instances arise from many diverse areas—many practical problems in AI planning [61], circuit testing [107], and software modeling [54]

J. Marques-Silva (✉)
University of Lisbon, Lisbon, Portugal
e-mail: jpms@ciencias.ulisboa.pt

S. Malik
Princeton University, Princeton, NJ, USA

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_9

can be formulated as SAT instances. This has motivated 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 [33], search [32], local search and random walk [98], Binary Decision Diagrams [25], Stålmarck's algorithm [100], and others. 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 [61] and FPGA routing [87], where instances are likely to be satisfiable and proving unsatisfiability is not required. However, for many other domains, including verification using model checking, the primary task is to prove unsatisfiability of the instances. Applications of SAT to model checking arise in bounded model checking [20], as well as interpolant—[83] and induction—[99] based approaches to unbounded model checking. For these, complete SAT solvers are a requirement.

In recent years search-based algorithms based on the well-known Davis–Logemann–Loveland algorithm [32] (sometimes referred to as 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 fifty 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) [31], POSIT [40], 2cl [112] and CSAT [36] among others. In the mid-1990s, Marques-Silva and Sakallah in the GRASP SAT solver [80, 81], and Bayardo and Schrag in the relsat SAT solver [14] proposed to augment the original DPLL algorithm with non-chronological backtracking and conflict-driven clause learning (CDCL). 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., [20, 54, 87]), 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 [118], Chaff [86], BerkMin [44] and more recently MiniSAT [38] and PicoSAT [19] which pay a lot of attention to optimizing various aspects of the DPLL algorithm. Some of these deal with efficient implementations of specific steps in the DPLL and CDCL, e.g., unit-propagation in SATO and Chaff, and others with more efficient search space pruning such as the locality-based search in Chaff. 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.

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 significant attention is focused on various aspects of the solver such as coding, data structures, choosing algorithms and heuristics for specific steps, and parameter tuning. In this chapter we chart the journey from the original basic DPLL framework through the introduction of efficient techniques within this framework culminating in state-of-the-art CDCL 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 consistently 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.

The chapter is organized as follows. Section 9.2 introduces the notation used throughout the chapter. Section 9.3 provides an overview of modern CDCL SAT solvers. Section 9.4 details the key techniques that are used in CDCL SAT solvers. Section 9.5 provides a brief overview of SAT-based problem solving, highlighting a number of problems of interest to model checking. Finally, Sect. 9.6 concludes the chapter.

9.2 Preliminaries

This section introduces the notation used in the remainder of the chapter. Standard propositional logic definitions are used throughout the chapter (e.g., [21, 62]). Boolean formulas are represented in calligraphic font, e.g., $\mathcal{F}, \mathcal{H}, \mathcal{S}, \mathcal{U}, \ldots$ Boolean variables are represented with lowercase letters from the start or the end of the alphabet, e.g., $a, b, c, \ldots, r, s, t, u, v, w, x, y, z$. Whenever necessary, subscripts can be used, e.g., $x_1, w_1, \ldots$ An atom is a Boolean variable. A literal is a variable x or its complement $\neg x$. For notational convenience, the complement of a variable x is represented as $\bar{x}$. A Boolean formula $\mathcal{F}$ is defined inductively over a set of propositional variables, with the standard logical connectives, $\neg$, $\wedge$, $\vee$, as follows:

1. An atom is a Boolean formula.
2. If $\mathcal{F}$ is a Boolean formula, then $(\neg\mathcal{F})$ is a Boolean formula. (When $\mathcal{F}$ represents an atom x, $\neg\mathcal{F}$ is represented by $\bar{x}$.)
3. If $\mathcal{F}$ and $\mathcal{G}$ are Boolean formulas, then $(\mathcal{F} \vee \mathcal{G})$ is a Boolean formula.
4. If $\mathcal{F}$ and $\mathcal{G}$ are Boolean formulas, then $(\mathcal{F} \wedge \mathcal{G})$ is a Boolean formula.

Similar definitions can be developed for the other logic connectives, $\rightarrow$ and $\leftrightarrow$. (The use of parentheses is not enforced, and standard binding rules apply (e.g., [62]), with parentheses being used only to clarify the presentation of formulas.) The variables of a Boolean formula $\mathcal{F}$ are represented by $\mathsf{var}(\mathcal{F})$. Set X is also used to refer to the set of variables of a formula, $X = \mathsf{var}(\mathcal{F})$. A clause c is a non-tautologous disjunction of literals. A term t is a non-contradictory conjunction of literals. Commonly used representations of Boolean formulas include conjunctive and disjunctive normal forms (resp. CNF and DNF). A CNF formula $\mathcal{F}$ is a conjunction of clauses. A DNF formula $\mathcal{F}$ is a disjunction of terms. CNF and DNF formulas can also be viewed as sets of sets of literals. The two representations will be used interchangeably throughout the chapter. In the remainder of the chapter, Boolean formulas are referred to as formulas, which includes CNF formulas and DNF formulas. The necessary qualification will be used when necessary.

Given a formula $\mathcal{F}$, a truth assignment ν is a map from the variables of $\mathcal{F}$ to $\{0, 1\}$, $\nu : \mathsf{var}(\mathcal{F}) \mapsto \{0, 1\}$.

Given a truth assignment ν, the value taken by a formula, denoted $\mathcal{F}^\nu$, is defined inductively as follows:

1. If x is a variable, $x^\nu = \nu(x)$.
2. If $\mathcal{F} = (\neg\mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 0 & \text{if } \mathcal{G}^\nu = 1 \\ 1 & \text{if } \mathcal{G}^\nu = 0. \end{cases}$$

3. If $\mathcal{F} = (\mathcal{E} \vee \mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 1 & \text{if } \mathcal{E}^\nu = 1 \text{ or } \mathcal{G}^\nu = 1 \\ 0 & \text{otherwise.} \end{cases}$$

4. If $\mathcal{F} = (\mathcal{E} \wedge \mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 1 & \text{if } \mathcal{E}^\nu = 1 \text{ and } \mathcal{G}^\nu = 1 \\ 0 & \text{otherwise.} \end{cases}$$

In some contexts, including search algorithms for the Boolean Satisfiability (SAT) problem, a truth assignment is relaxed to be partial, i.e., not all variables are assigned a truth value. A truth assignment is complete if the map is total; otherwise it is partial. For a partial truth assignment, if $\nu(x)$ is not specified, then we write $\nu(x) = u$.

For a CNF formula $\mathcal{F}$, let ν be a truth assignment. A clause c is *satisfied* if there exists a literal $l \in c$, such that $l^\nu = 1$. If all literals of c take value 0, then the clause is *falsified*. If all literals but one are assigned value 0, and the remaining one is unassigned, then the clause is *unit*. Finally, a clause is unresolved if it is neither falsified, nor satisfied, nor unit. A CNF formula is satisfied if all clauses are satisfied, and falsified if at least one clause is falsified.

A truth assignment is *satisfying* for $\mathcal{F}$ (or simply a satisfying truth assignment) if $\mathcal{F}^\nu = 1$. A formula $\mathcal{F}$ is *satisfiable* if it has a satisfying truth assignment; otherwise it is *unsatisfiable*. If a formula $\mathcal{F}$ is satisfiable, we write $\mathcal{F} \nvDash \bot$. If a formula $\mathcal{F}$ is unsatisfiable, we write $\mathcal{F} \vDash \bot$.

Definition 1 (Boolean Satisfiability (SAT)) Given a formula $\mathcal{F}$, the decision problem SAT consists of deciding whether $\mathcal{F}$ is satisfiable.

CDCL SAT solvers, but also DPLL SAT solvers, implement some form of backtracking search. Both CDCL and DPLL SAT solvers branch on variables; these are referred to as *decision variables*.

A key procedure in SAT solvers is the *unit clause rule* [33]: if a clause is unit, then its sole unassigned literal must be assigned value 1 for the clause to be satisfied. The iterated application of the unit clause rule is referred to as *unit propagation* or *Boolean constraint propagation* (BCP) [117]. In modern CDCL solvers, as in most implementations of DPLL, logical consequences are derived with unit propagation.

Unit propagation is applied after each branching step (and also during preprocessing[1]), and is used for identifying variables that must be assigned a specific Boolean value. If a falsified clause is identified, a *conflict* condition is declared, and the algorithm backtracks.

In CDCL SAT solvers, each variable x is characterized by a number of properties, including the *value*, the *antecedent clause* (or just *antecedent*) and the *decision level*, denoted respectively by $\nu(x) \in \{0, u, 1\}$, $\alpha(x) \in \mathcal{F} \cup \{\text{NIL}\}$, and $\delta(x) \in \{-1, 0, 1, \ldots, |X|\}$. A variable x that is assigned a value as the result of applying the unit clause rule is said to be *implied*. The unit clause c used for implying variable x is said to be the antecedent of x, $\alpha(x) = c$. For variables that are decision variables or are unassigned, the antecedent is NIL. Hence, antecedents are only defined for variables whose value is implied by other assignments. The decision level of a variable x denotes the depth of the decision tree at which the variable is assigned a value in $\{0, 1\}$. The decision level for an unassigned variable x is -1, $\delta(x) = -1$. The decision level associated with variables used for branching steps (i.e., *decision assignments*) is specified by the search process, and denotes the current depth of the *decision stack*. The decision stack represents the sequence of branched-upon variables. Hence, a variable x associated with a decision assignment is characterized by having $\alpha(x) = \text{NIL}$ and $\delta(x) > 0$. When describing and analyzing SAT solvers, *implication graphs* [80, 81] are used to graphically depict the application of unit propagation at each decision level, as a consequence of each branching decision. Each node in the implication graph shows a literal, with the incoming edges to each literal identifying the antecedent of the assignment. If a falsified clause is identified by unit propagation, this is marked in the implication graph with a special node $\perp$. The implication graph can be viewed as a graphical representation of the relationship between implied variables and their antecendents.

Figure 1 exemplifies the implication graphs considered in this chapter. This example also illustrates the above definitions. With the exception of decision level 0, a decision literal is associated with each decision level. For example, for decision level 1, the decision literal is w, denoting that w is assigned value 1. For simplicity all examples shown just use positive literals (i.e., variables are always decided or implied value 1). Given the implication graph, the antecedent of a given implied assignment can be inferred from the incoming edges. For example, b is assigned value 1 because a and x are assigned value 1. Hence, the antecedent of b is $(\bar{x} \vee \bar{a} \vee b)$.

A standard operation associated with Boolean formulas is *resolution* [33, 94]. Given clauses $C_1 = (x \vee A)$ and $C_2 = (\bar{x} \vee B)$, where A and B are disjunctions of literals without complemented literals, the resolution of C_1 and C_2 is $C_3 = (A \vee B)$. As shown in Sect. 9.4, resolution serves to explain a wide range of techniques used in modern SAT solvers, including CDCL SAT solvers. For example, unit propagation can be explained with resolution operations and, as illustrated in Sect. 9.4.1, clause learning can also be explained as a sequence of resolution operations. Moreover, resolution is also associated with a number of complete proof systems for SAT (e.g., [62, 111]).

[1]Preprocessing serves to simplify Boolean formulas and is briefly covered in Sect. 9.4.7.

$$\mathcal{F} = (r) \wedge (\bar{r} \vee s) \wedge (\bar{w} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge$$
$$(\bar{y} \vee \bar{z} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d)$$

$$\begin{tikzcd}
\text{Level} & \text{Dec.} & \text{Unit Prop.} & \\
0 & \emptyset & r \arrow[r] & s \\
1 & w \arrow[r] & a \arrow[d] & \\
2 & x \arrow[r] & b \arrow[ddr, bend left] & \\
3 & y \arrow[dr, bend left] & & \\
4 & z \arrow[r] & c \arrow[r] & d
\end{tikzcd}$$

Fig. 1 Example of notation and unit propagation

Modern SAT solvers typically accept CNF formulas [78]. This is due to the inexpensive deduction provided by unit propagation. Procedures for CNF-encoding (or clausifying) arbitrary Boolean formulas are well-known (e.g., [90, 110]).

9.3 CDCL SAT Solvers: Organization

This section provides a high-level description of modern CDCL SAT solvers. Afterwards, Sect. 9.4 details the most important algorithmic techniques associated with CDCL SAT solvers, namely conflict-driven clause learning [80, 81], unique implication points [80, 81], learned clause minimization [105], lazy data structures [86], search restarts [11, 45] and lightweight branching heuristics [86].

Algorithm 1 shows the standard organization of a CDCL SAT solver, which essentially follows the organization of DPLL. With respect to DPLL, the main differences are the call to function ConflictAnalysis each time a conflict is identified, and the call to Backtrack when backtracking takes place. Moreover, the Backtrack procedure allows for backtracking non-chronologically.

In addition to the main CDCL function, the following auxiliary functions are used:

- UnitPropagation consists of the iterated application of the unit clause rule. If a falsified clause is identified, then a conflict indication is returned.
- PickBranchingVariable consists of selecting a variable and assigning it a value.
- ConflictAnalysis consists of analyzing the most recent conflict and learning a new clause from the conflict. The organization of this procedure is described in Sect. 9.4.1.
- Backtrack backtracks to the decision level computed by ConflictAnalysis.
- AllVariablesAssigned tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable.

Algorithm 1 Typical CDCL algorithm

```
CDCL($\mathcal{F}$, $\nu$)
1  if (UnitPropagation($\mathcal{F}$, $\nu$) == CONFLICT)
2     then return UNSAT
3  $dl \leftarrow 0$                         ▷ Decision level
4  while (not AllVariablesAssigned($\mathcal{F}$, $\nu$))
5       do $(x, v)$ = PickBranchingVariable($\mathcal{F}$, $\nu$)
6          $dl \leftarrow dl + 1$            ▷ Increment decision level due to new decision
7          $\nu \leftarrow \nu \cup \{(x, v)\}$
8          if (UnitPropagation($\mathcal{F}$, $\nu$) == CONFLICT)
9             then $\beta$ = ConflictAnalysis($\mathcal{F}$, $\nu$)
10                 if ($\beta < 0$)
11                    then return UNSAT
12                    else Backtrack($\mathcal{F}$, $\nu$, $\beta$)
13                         $dl \leftarrow \beta$      ▷ Decrement decision level due to
                                                    backtracking
14 return SAT
```

An alternative criterion to stop execution of the algorithm is to check whether all clauses are satisfied. However, in modern SAT solvers that use lazy data structures, clause state cannot be maintained accurately, and so the termination criterion must be whether all variables are assigned. Thus, in this case the algorithm provides a complete assignment.

Arguments to the auxiliary functions are assumed to be passed by reference. Hence, $\mathcal{F}$ and ν are supposed to be modified during execution of the auxiliary functions.

The typical CDCL algorithm shown does not account for a few often-used techniques, namely search restarts [11, 45] and implementation of clause deletion policies [44]. Search restarts cause the algorithm to restart itself. However, past search history is not erased, for example previously learnt clauses are kept. Clause deletion policies are used to decide learned clauses that can be deleted based on their expected future utility. Clause deletion allows the memory usage of the SAT solver to be kept under control.

9.4 CDCL SAT Solvers

This section reviews the techniques that are common to CDCL SAT solvers. These techniques can be organized as follows:

1. Conflict-driven clause learning [80, 81].
2. Unique implication points [80, 81].
3. Learned clause minimization [105].

Algorithm 2 Main steps of conflict analysis procedure

```
CONFLICTANALYSIS($\mathcal{F}$, $\nu$)
1  Start at node $\perp$
2  Recursively visit literals of antecedents assigned at current decision level
3  Record complement of antecendent literals assigned at lower decision levels
4  Record complement of branching literal
5  Create clause with recorded literals
6  return largest decision level of recorded literals other than the current level
```

4. Lazy data structures [86].
5. Search restarts [11, 45].
6. Lightweight branching heuristics [86].
7. Additional techniques [7, 44, 89].

9.4.1 Clause Learning and Non-chronological Backtracking

Learning from conflicts has been extensively studied in a number of areas since the 1970s (e.g., [106]). In some contexts, learning from conflicts was shown to be ineffective, both in theory and in practice [9, 115]. Clause learning in SAT solvers [80, 81] is inspired by this earlier work on learning from conflicts, but exhibits important differences. The most important aspect is that clause learning exploits the sequence of unit propagation steps that produces the conflict. In addition, clause learning in SAT solvers exploits UIPs (see Sect. 9.4.2). The original ideas of clause learning in SAT solvers were proposed in the GRASP SAT solver [72, 80, 81]. A recent alternative formalization of clause learning can be found in [78]. This section overviews clause learning by summarizing the main steps and illustrating how these are applied to a simple example.

As the CDCL algorithm is executed, if a falsified clause is identified, conflict analysis is used to create a clause that explains and prevents the same conflict from re-occurring. Algorithm 2 summarizes the main steps of the conflict analysis (and learning) procedure. The input arguments are the CNF formula, and the current set of assignments. Literals implied at the current decision level are traversed, starting from the $\perp$ vertex (which represents the falsified clause). For each traversed literal, the literals in the antecedent are analyzed. A literal assigned at a decision level lower than the current one has its complemented literal recorded, whereas a literal assigned at the current decision level is scheduled to be traversed. The process is repeated until the branching variable for the current decision level is visited.

Figure 2 shows a simple example of unit propagation yielding a conflict. The implication graph summarizes how unit propagation produces the conflict. Algorithm 2 is executed on the implication graph, starting from node $\perp$. Literals a, b, and z are visited, since all are assigned at decision level 3. The recorded literals are $\bar{x}$ and $\bar{z}$. Thus, the created clause is $(\bar{x} \vee \bar{z})$. These steps are shown in Fig. 3.

$$\mathcal{F} = (\bar{x} \vee \bar{z} \vee a) \wedge (\bar{z} \vee b) \wedge (\bar{a} \vee \bar{b})$$

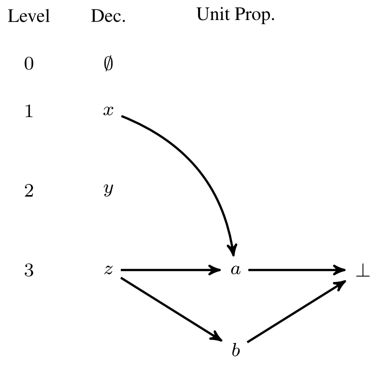


Fig. 2 Clause learning: (a) example formula and (b) conflict after unit propagation

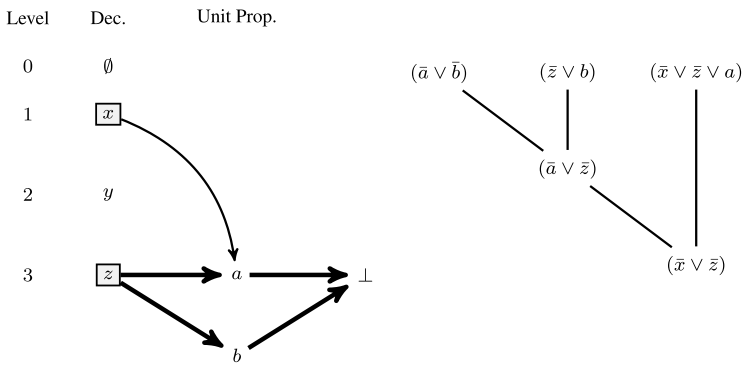


Fig. 3 Clause learning: creating a new clause

Traversed edges are marked with thick lines. Each literal for which the complement is recorded is highlighted and shown inside a box. Moreover, the derivation of the learned clause is formally explained by the application of a sequence of (selected) resolution steps. Hence, clause learning can be viewed as a way to decide which clauses to learn by selective resolution steps. Figure 4 also shows the result after backtracking. The backtrack step shown is the one proposed in [86], which differs somewhat from the backtrack step originally associated with clause learning in the GRASP SAT solver [80, 81]. The GRASP SAT solver delayed backtracking until both assignments had been considered for the branching variable. This would avoid possibly unnecessary (and, in the case of GRASP, expensive) backtracking.

A number of researchers have investigated ways to improve the basic clause learning procedure outlined above (e.g., [6]). Nevertheless, most state-of-the-art SAT solvers implement the basic clause learning procedure, first proposed in GRASP [80, 81], with the backtracking step used in Chaff, but improved with learned clause minimization (which is described in Sect. 9.4.3). Recent work ad-

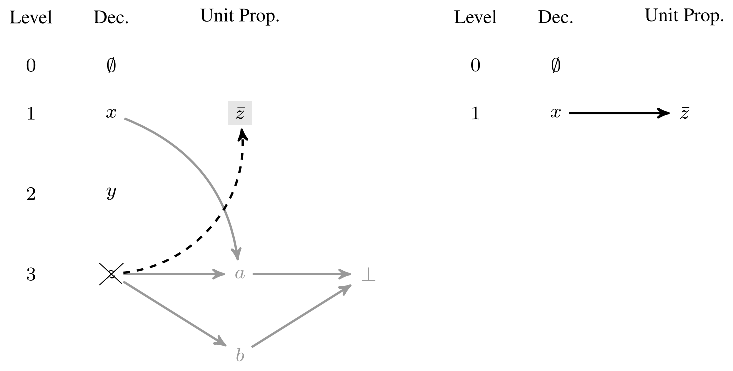


Fig. 4 Clause learning: after backtracking

dresses techniques to decide which clauses are expected to be of interest for the subsequent search [7].

9.4.2 Unique Implication Points

A key aspect of clause learning in SAT solvers is *Unique Implication Points* (UIPs). If unit propagation due to a branching decision yields a conflict, then any dominator [109] of the conflict node with respect to the branching decision is a UIP [80, 81]. UIPs can be related with *failure-driven assertions* [79], used in the context of circuit testing, and mimic, at the logic level, the notion of unique sensitization points (USPs) also used in testing [42]. UIPs serve a number of purposes in CDCL SAT solvers. First, UIPs allow learning of smaller clauses. Second, UIPs allow learning of multiple clauses. The clause learning procedure outlined in Algorithm 2 can be modified to stop when the first dominator is identified. The intuitive justification for this is that assigning the literal associated with the UIP suffices to reproduce the conflict. Hence, the clause learning procedure can terminate by recording the complement of the UIP literal.

Figure 5 illustrates the use of UIPs in clause learning. For this example, without the identification of UIPs, the learned clause would be $(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$. This is shown in Fig. 6, where the clause is learned following the steps outlined earlier. However, if clause learning stops at the first UIP, then the learned clause becomes $(\bar{w} \vee \bar{x} \vee \bar{a})$. Observe that stopping at the first UIP essentially consists of performing fewer resolution steps, i.e., the clause learned by stopping at the first UIP is already present in the resolution steps used to derive the learned clause without stopping at the first UIP.

Moreover, observe that, for this concrete example, the learned clause is not only smaller, but induces backtracking to a lower decision level. A straightforward observation is that clauses learned by stopping learning at the first UIP result in backtracking decision levels that are no larger than the decision levels of clauses learned

$$\mathcal{F} = (\bar{y} \vee \bar{z} \vee a) \wedge (\bar{x} \vee \bar{a} \vee b) \wedge \\ (\bar{w} \vee \bar{a} \vee c) \wedge (\bar{b} \vee \bar{c})$$

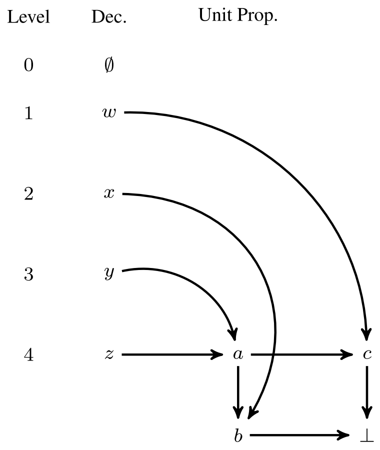


Fig. 5 Unique implication points: (a) example formula and (b) conflict after unit propagation

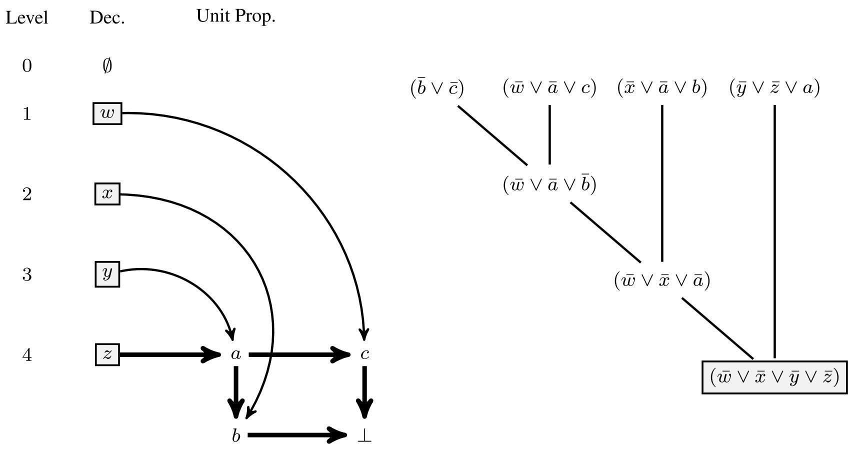
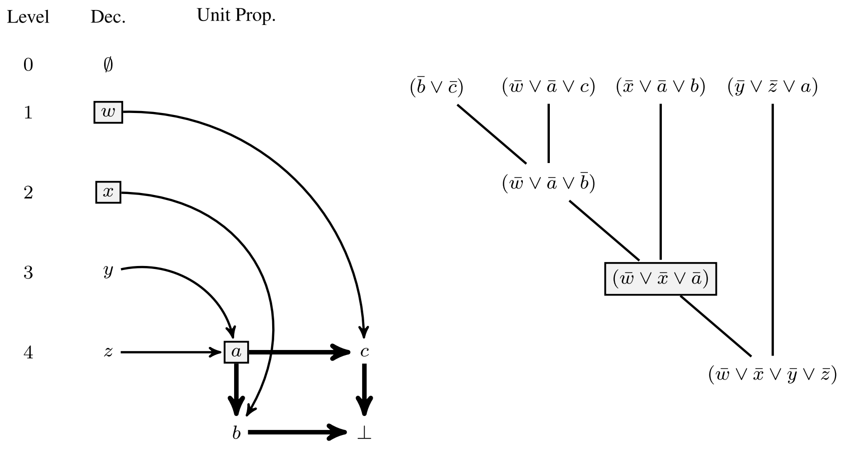

Level
Dec.
Unit Prop.
0
∅
1
w
2
x
3
y
4
z
a
c
b
⊥
$(\bar{b} \vee \bar{c})$
$(\bar{w} \vee \bar{a} \vee c)$
$(\bar{x} \vee \bar{a} \vee b)$
$(\bar{y} \vee \bar{z} \vee a)$
$(\bar{w} \vee \bar{a} \vee \bar{b})$
$(\bar{w} \vee \bar{x} \vee \bar{a})$
$(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$

Fig. 6 Clause learning without UIPs

by stopping at the decision literal. A slightly more detailed characterization of this property can be found in [6].

Although the modern usage of UIPs is based on stopping clause learning at the first UIP, the original approach was to learn clauses at every UIP [80, 81]. Recent results, obtained on problem instances from the SAT competitions, suggest that learning clauses at multiple UIPs can improve SAT solver performance [96]. An example of clause learning at multiple UIPs is shown in Fig. 8. As shown in Fig. 9a, conflict analysis by stopping at the first UIP produces the learned clause $(\bar{w} \vee \bar{y} \vee \bar{a})$. However, it is possible to continue learning clauses at each additional UIP. For the example in Fig. 8, z is also a UIP (it is actually the UIP corresponding to the decision variable). Observe that ($x = 1$ and) $z = 1$ implies $a = 1$, and so $a = 0$ implies $z = 0$.



Level
Dec.
Unit Prop.
0
∅
1
$\boxed{w}$
2
$\boxed{x}$
3
y
4
z
$\boxed{a}$
c
b
$\perp$
$(\bar{b} \vee \bar{c})$
$(\bar{w} \vee \bar{a} \vee c)$
$(\bar{x} \vee \bar{a} \vee b)$
$(\bar{y} \vee \bar{z} \vee a)$
$(\bar{w} \vee \bar{a} \vee \bar{b})$
$\boxed{(\bar{w} \vee \bar{x} \vee \bar{a})}$
$(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$

Fig. 7 Clause learning with UIPs

$$\begin{aligned}\mathcal{F} = {} & (\bar{z} \vee r) \wedge (\bar{z} \vee s) \wedge \\ & (\bar{x} \vee \bar{r} \vee \bar{s} \vee a) \wedge (\bar{b} \vee \bar{c}) \wedge \\ & (\bar{y} \vee \bar{a} \vee b) \wedge (\bar{w} \vee \bar{a} \vee c)\end{aligned}$$

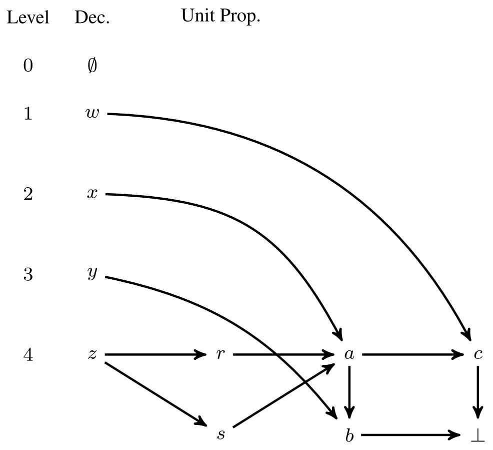


Fig. 8 Multiple UIPs: (a) example formula and (b) conflict after unit propagation

However, this information is not obtained by unit propagation, i.e., $a = 0$ does *not* lead to $z = 0$. Nevertheless, by noting that z is a UIP, the following clause is learned: $(\bar{z} \vee \bar{x} \vee a)$. This is illustrated in Fig. 9b. With this additional clause added to the formula, $a = 0$ now implies $z = 0$ whenever $x = 1$. The clauses obtained by clause learning at multiple UIPs are inspired by, but generalize, the concept of global implications first studied in the area of circuit testing [97].

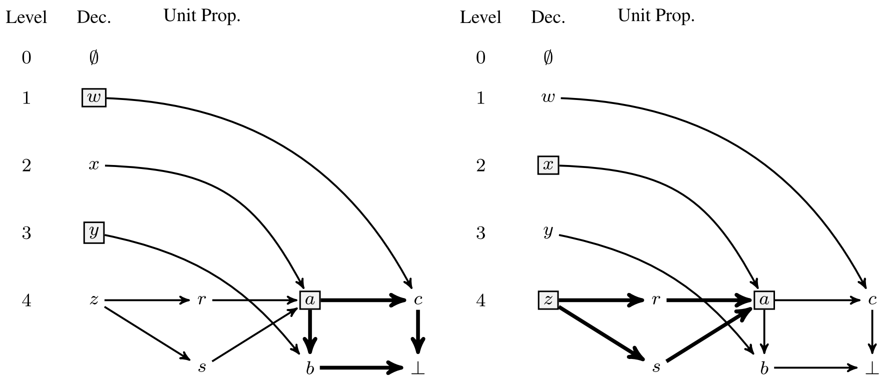


Fig. 9 Multiple UIPs: (a) first UIP clause; and (b) second UIP clause

$$\begin{aligned}\mathcal{F} = {} & (\bar{x} \vee b) \wedge (\bar{z} \vee \bar{b} \vee c) \wedge \\ & (\bar{x} \vee \bar{y} \vee \bar{z} \vee a) \wedge (\bar{a} \vee \bar{c})\end{aligned}$$

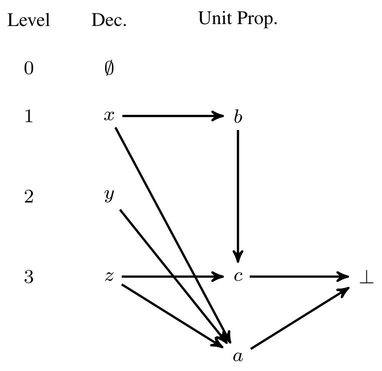
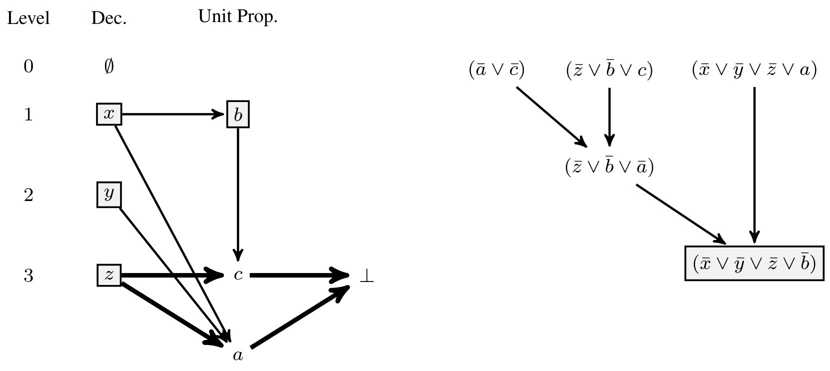


Fig. 10 Learned clause minimization: (a) example formula; (b) conflict after unit propagation

9.4.3 Learned Clause Minimization

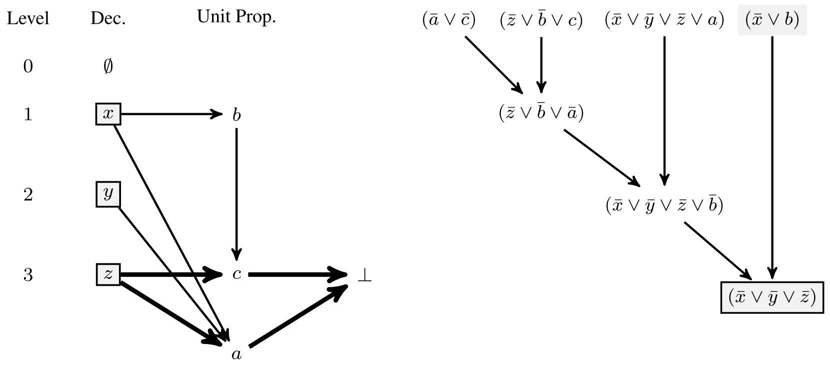
The basic clause learning procedure has not changed significantly since the mid-1990s [80, 81]. However, recent SAT solvers exploit a key optimization step after clause learning: *learned clause minimization* [105]. In the mid-2000s, researchers noticed that learned clauses exhibit important redundancies, and that these can be removed with simple procedures. The performance gains obtained with learned clause minimization justify the inclusion of this technique in most modern SAT solvers.

Let $C_1 = (x \vee A)$ and $C_2 = (\bar{x} \vee A \vee B)$ be clauses of $\mathcal{F}$, where A and B are disjunctions of literals. Resolution between C_1 and C_2 produces the clause $C_3 = (A \vee B)$ which subsumes C_2. If C_3 is added to $\mathcal{F}$, then C_2 can be removed from $\mathcal{F}$, since it is subsumed by C_3. This form of resolution is called *self-subsuming resolution* [105]. One clause minimization procedure consists of the iterative application of self-subsuming resolution between a learned clause c and the antecedents of the literals in c [105]. Figure 10 shows an example of clause minimization by self-subsuming resolution. Clause learning without clause minimization, shown in



Level
Dec.
Unit Prop.
0
∅
1
2
3
x
y
z
b
c
a
$\perp$
$(\bar{a} \vee \bar{c})$
$(\bar{z} \vee \bar{b} \vee c)$
$(\bar{x} \vee \bar{y} \vee \bar{z} \vee a)$
$(\bar{z} \vee \bar{b} \vee \bar{a})$
$(\bar{x} \vee \bar{y} \vee \bar{z} \vee \bar{b})$

Fig. 11 Clause learning without minimization



Level
Dec.
Unit Prop.
0
∅
1
2
3
x
y
z
b
c
a
$\perp$
$(\bar{a} \vee \bar{c})$
$(\bar{z} \vee \bar{b} \vee c)$
$(\bar{x} \vee \bar{y} \vee \bar{z} \vee a)$
$(\bar{x} \vee b)$
$(\bar{z} \vee \bar{b} \vee \bar{a})$
$(\bar{x} \vee \bar{y} \vee \bar{z} \vee \bar{b})$
$(\bar{x} \vee \bar{y} \vee \bar{z})$

Fig. 12 Minimization with self-subsuming resolution

Fig. 11, yields the learned clause $C_l = (\bar{x} \vee \bar{y} \vee \bar{z} \vee \bar{b})$. However, clause learning followed by self-subsuming resolution between C_l and the antecedent of b yields the clause $C_l' = (\bar{x} \vee \bar{y} \vee \bar{z})$, as shown in Fig. 12. Observe that, in contrast with UIPs, self-subsuming resolution steps are resolution steps which are appended to the resolution derivation to generate the final minimized learned clause.

In practice, self-subsuming resolution is often not enough to effectively minimize learned clauses. An alternative is the so-called *recursive minimization* procedure [105], which is summarized in Algorithm 3. Figure 13 shows an example of applying recursive clause minimization. As shown in Fig. 14, clause learning without minimization yields clause $(\bar{w} \vee \bar{x} \vee \bar{c})$. For this example, self-subsuming resolution cannot be applied, because resolution operations make the resulting clause larger. However, the recursive clause minimization procedure can be used to prove that literal $\bar{c}$ can be dropped from the clause. As shown in Fig. 14b, the traversal from vertex c solely reaches marked vertex w. Hence, the literal $\bar{c}$ can be dropped from the learned clause, and so the final clause becomes $(\bar{w} \vee \bar{x})$.

Algorithm 3 Main steps of recursive clause minimization procedure

```
RecursiveClauseMinimization($c$)
1  Mark literals in $c$
2  Implied literals in $c$ are flagged as candidates for removal
3  foreach candidate literal $l$ in $c$
4      do Traverse implication graph starting from antecedent of $l$
5         Stop at decision literals or marked literals
6         if Non-marked literal visited
7            then Keep literal $l$ in $c$
8            else Drop literal $l$ from $c$
9  return $c$
```

$$\begin{aligned}\mathcal{F} = {} & (\bar{w} \lor a) \land (\bar{w} \lor b) \land \\ & (\bar{a} \lor \bar{b} \lor c) \land (\bar{w} \lor \bar{x} \lor d) \land \\ & (\bar{x} \lor \bar{c} \lor e) \land (\bar{e} \lor \bar{d})\end{aligned}$$

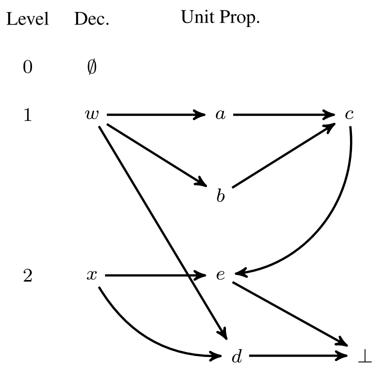


Fig. 13 Learned clause minimization: (a) example formula; (b) conflict after unit propagation

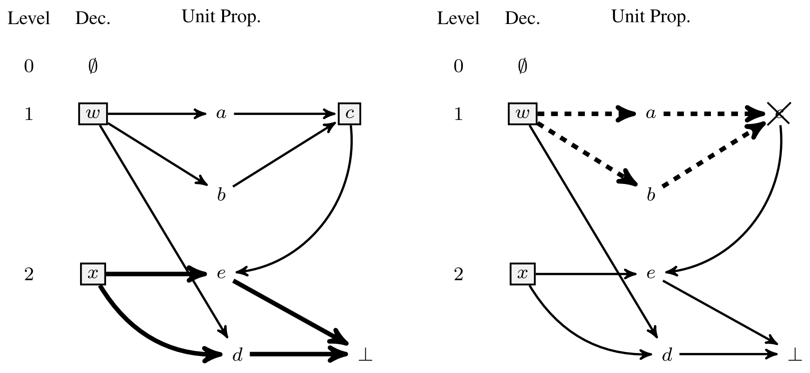
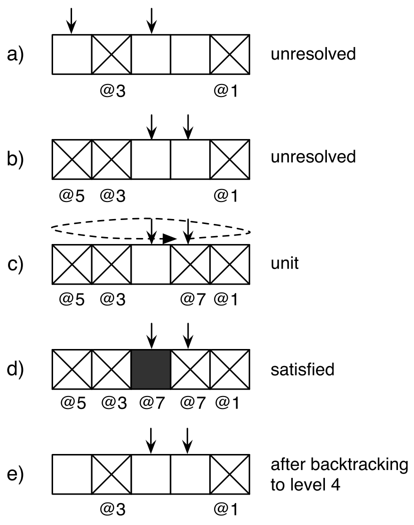


Fig. 14 Clause learning: (a) no minimization; and (b) recursive minimization

9.4.4 Lazy Data Structures

Until the early 2000s most DPLL/CDCL SAT solvers used adjacency lists as the underlying data structure for clause representation [70], with the exception being the

Fig. 15 Operation of watched literals



head-tail representation proposed in SATO [119]. Adjacency lists require L references from literals to clauses, where L denotes the total number of literals. This can become an issue when learning many (possibly large) clauses. The head-tail representation requires between $2 \times C$ and L references from literals to clauses [70, 78], where C denotes the number of clauses. Although more efficient in practice than adjacency lists, the head-tail representation causes overhead when backtracking, besides requiring a varying number of references.

One of the main contributions of the Chaff SAT solver was the use of a new (lazy) data structure, the *watched literals* data structure. The watched literals data structure has several important advantages. First, for each clause only two references from literals to the clause are required. This results in $2 \times C$ references in total. Also, when backtracking, no bookkeeping is required. This provides significant performance gains over the other data structures. As a result, watched literals have become the de facto standard in the implementation of modern SAT solvers (e.g., [19, 38]). Observe that the lowest number of references for each clause is 2, since one must be able to decide when the clause is unit so that unit propagation can be used to assign a value to some variable. Figure 15 illustrates the operation of the watched literals data structure, being adapted from [78]. The example considers a single clause with 5 literals, with the arrows showing the currently watched literals. The current decision level is either 3 or 4, and the clause has 2 literals already assigned value 0 (shown crossed out in the figure). At decision level 5, one of the watched literals is assigned value 0. This requires the algorithm to find another literal to watch, and so the reference is updated. At decision level 7, another watched literal is assigned value 0. In this case, all literals are visited, trying to find a literal that is still unassigned and not watched. In this case none exists, i.e., all literals but one are assigned value 0 and the remaining unassigned literal is already watched. Hence, the clause is declared unit. As a result, the only unassigned literal is assigned value 1 (shown as a black box in the figure), so that the clause becomes satisfied.

Afterwards, the algorithm backtracks to decision level 4. As indicated earlier, there is no need to update the references. Thus, when backtracking, the watched literal data structure requires no bookkeeping.

9.4.5 Search Restarts

Another standard ingredient in modern SAT solvers is search restarts [45]. Research in the late 1990s showed that DPLL SAT solvers exhibit heavy-tail behavior on satisfiable problem instances when the branching heuristic is randomized [45]. This means that the run times of DPLL SAT solvers can exhibit large variations for the same satisfiable instance, and that large run times can happen with non-negligible probability. This observation motivated the proposal of *rapid randomized restarts*, i.e., to restart the search after a fixed (or alternatively increasing) number of conflicts. The increase in the number of conflicts is one possible technique to guarantee that the SAT algorithm is still complete when search restarts are implemented; another is to keep *all* learned clauses [68]. Later work [11] showed that search restarts were also very effective for CDCL SAT solvers, and for solving unsatisfiable problem instances. These conclusions were further substantiated by the implementation of search restarts in the Chaff SAT solver [86].

As with the techniques described in earlier sections, search restarts are commonly used in modern CDCL SAT solvers [19, 38, 78]. In recent years, a number of works have studied different restart policies, including [7, 19, 52, 103].

9.4.6 Lightweight Branching Heuristics

Modern CDCL SAT solvers also exploit so-called lightweight branching heuristics, most notably the VSIDS branching heuristic [86]. The previous generation of branching heuristics [73] maintained counts of assigned literals in each clause. This incurs a significant overhead. For example, in the GRASP SAT solver, branching could account for more than two thirds of the run time [70]. In contrast, lightweight branching heuristics use solely information from conflicts to decide which variables to branch upon. Hence, static or dynamic literal counts are not required. Variables that are involved in more conflicts are more likely to be used for branching than variables not involved in conflicts. This is achieved by associating a metric with each variable, which is incremented for variables involved in conflicts. On average, the most recent conflicts are more relevant than earlier conflicts, since these may no longer be useful for the current state of the search. As a result, VSIDS divides the variable metrics by a constant after a fixed number of conflicts. Besides the low overhead of this heuristic, it also results in what is called locality-based search. Since the variables occurring in recent conflicts are weighted more heavily, the algorithm is biased towards branching on these variables. Thus, the search focuses on the sub-space of recent conflicts, effectively pruning this sub-space before moving on to other sub-spaces. While only intuitively understood, this has a very significant impact on the size of the search space explored and is credited with the speed-up of this generation of SAT solvers. As with the techniques described in earlier subsections, the VSIDS branching heuristic has become a de facto standard in modern CDCL SAT solvers.

9.4.7 Additional Techniques and Recent Trends

This section reviews a few other techniques that are found in modern CDCL SAT solvers. One key issue with CDCL SAT solvers is that the number of learned clauses can become too large. As a result, researchers have developed different solutions for this problem since the mid-1990s. Original solutions were based on restricting the size of learned clauses [14, 80, 81]. More recent work proposes the use of different metrics to decide which clauses to delete [7, 44]. Earlier work considered activity heuristics [44], i.e., if a clause is not used for unit propagation, then it can be marked for deletion. More recent work gives preference to deleting clauses whose literals are distributed by more decision levels [7].

The main change to the organization of branching is the use of *phase saving* [89], i.e., the value of each assigned literal is saved when backtracking takes place. Afterwards, this saved value is reused when that literal is branched upon.

Formula preprocessing has been studied extensively [24, 69, 74]. Recent work has shown that specific forms of preprocessing are effective [37, 57]. Among the many techniques that have been proposed, the most widely used include variable elimination, blocked clause elimination and elimination of subsumed clauses. Moreover, preprocessing techniques have been integrated within SAT solvers, under the general framework of *inprocessing* [58].

Additional promising research directions include algorithm portfolios for SAT [96, 116] and parallel algorithms for SAT [47–49].

9.5 SAT-Based Problem Solving

The importance of SAT solvers is demonstrated by the many problem-solving uses of SAT. This section overviews the different ways in which SAT solvers can be used for solving different problems.

The standard use of SAT solvers is as an engine for solving decision problems, i.e., requiring a yes/no answer. A large number of practical applications of SAT also involve iterative SAT solving, i.e., the problem to be solved requires calling a SAT solver a number of times. Clearly, the number of calls to the SAT solver is paramount in the overall efficiency.

In some cases, the number of calls to the SAT solver is polynomial in the size of the problem instance, but in some other cases the number of calls to the SAT solver is exponential in the worst case.

9.5.1 Incremental SAT

A key issue with iterative use of SAT solvers is how to communicate minimal changes in the formula to the SAT solver and, rather more importantly, how to reuse the learned clauses from previous SAT solver calls. One alternative is to communicate the complete CNF formula each time the SAT solver is to be called. This approach is often referred to as non-incremental, and reuse of learned clauses is not used. Another alternative is to communicate to the SAT solver only the clauses that should be discarded (or deactivated) and the new clauses that should be considered (or activated). This alternative is referred to as incremental, and its use in applications based on iterative SAT solving is now common. The essential ideas for incremental SAT solving are summarized below.

Most modern SAT solvers achieve these goals by using *assumptions* [38]. Clauses in the SAT solver are associated with a new assumption variable. Then, assumption variables are used in each SAT solver call to activate/deactivate clauses. The use of assumptions has important advantages and significant disadvantages. First, any learned clause will keep a record of the clauses explaining its derivation. Thus, activation (resp. deactivation) of assumption variables immediately activates (resp. deactivates) learned clauses that are usable (resp. unusable) in the next SAT solver call.

Another technique to implement incremental SAT, and so to allow reuse of learned clauses, is to use some proof-tracing mechanism [2, 19] (which includes representation of resolution proofs) [19].

Both approaches listed have advantages and disadvantages. Nevertheless, the use of assumptions is more widespread in published work.

9.5.2 Unsatisfiable Cores

In many SAT applications, including model checking, SAT solvers are expected to produce unsatisfiable cores [120], i.e., a subset of the original subformula which was used to prove unsatisfiability. Alternatively, a SAT solver can produce a resolution proof [120]. Unsatisfiable cores find a wide range of applications, including model checking [22], debugging specifications [101], and abstraction refinement [15]. Resolution proofs also find different applications, e.g., in computing interpolants [84].

Two main alternatives exist for computing unsatisfiable cores. The original approach consists of tracing the process of clause learning in CDCL SAT solvers, e.g., by writing an explanation for each learned clause to disk (or keeping it in memory in a separate data structure). Examples of variants of this approach include [2, 19, 120]. A widely used alternative is based on the use of assumption variables (see previous section). When learning clauses, all assumption variables associated with the clauses used for explaining a learned clause are added to the learned clause. Thus, when the SAT solver terminates, instead of producing the empty clause, it produces a clause containing the list of assumption variables of all clauses involved in proving the instance unsatisfiable, i.e., an unsatisfiable core.

9.5.3 CNF Encodings

In most uses of SAT, problems are not initially represented in CNF, e.g., [108]. As a result, a large body of research has been dedicated to encoding richer domains into CNF. Concrete examples include Satisfiability Modulo Theories (SMT), Constraint Satisfaction Problems (CSP), Answer Set Programming (ASP), but also simple extensions of propositional logic, that include non-clausal and pseudo-Boolean (PB) constraints.[2]

Encodings to CNF often address two key aspects. First, the size of the resulting CNF formula, namely whether the size of the encoding is polynomial in the original problem representation. Second, whether the CNF encoding preserves arc-consistency (e.g., see [1, 8, 92]), i.e., whether unit propagation suffices to (i) identify partial assignments that cannot be extended to a satisfying assignment; and (ii) identify any necessary assignments.

A number of ways exist to encode SMT into SAT. An up-to-date review is provided in [63]. Similarly to SMT, there are a number of ways to encode CSP into SAT. An overview of CSP to SAT encodings is provided in [92]. Like with SMT and CSP, there are also different ways to encode ASP into SAT. A recent account is provided in [55].

In many model-checking applications, instances of SAT are naturally non-clausal (e.g., interpolants in interpolant-based model checking [84]). As a result, mechanisms for encoding non-clausal formulas into clausal form have been developed (e.g., [90, 110]). A recent survey of these encodings is provided in [92].

For many practical applications, the domain variables are Boolean and the goal is to encode a pseudo-Boolean (PB) constraint of the general form:

$$\sum_{j=1}^{n} a_j x_j \bowtie b \tag{1}$$

where $\bowtie \in \{<, \leq, =, \geq, >\}$, $a_j \geq 0$, with $j \in \{1, \ldots, n\}$, $b \geq 0$, and x_j are propositional. For analyzing the size of the encodings, a_M denotes the value of the largest coefficient in (1).

A number of special cases of (1) have been extensively studied in the past. These include cardinality constraints of the form AtMost$k$, AtLeast$k$, and Equals$k$:

$$\sum_{j=1}^{n} x_j \bowtie k \tag{2}$$

Of these, constraints of the form AtMost1 have also been extensively studied [92]. (Observe that an AtLeast1 constraint can be trivially encoded with a clause, and so an Equals1 constraint can be encoded with an AtLeast1 and an AtMost1 constraint.)

There is a vast body of work on encoding PB constraints, cardinality constraints and AtMost1/Equals1 constraints [92]. Table 1 shows examples of CNF encodings.

[2]See [21] and references therein.

Table 1 Examples of CNF Encodings

Type	Encoding	# Clauses	Arc-Consistency	Reference
Pseudo-Boolean	Operational	linear	No	[113]
	BDD	exponential	Yes	[39]
	GPWE	$\mathcal{O}(n^3 \log(n) \log(a_{\mathrm{M}}))$	Yes	[8]
	GPWE*	$\mathcal{O}(n^3 \log(a_{\mathrm{M}}))$	Yes	[1]
Cardinality	BDD	$\mathcal{O}(n\,k)$	Yes	[39]
	Seq. Counter	$\mathcal{O}(n\,k)$	Yes	[102]
	Sort. Networks	$\mathcal{O}(n\,\log^2 n)$	Yes	[13, 39]
	Card. Networks	$\mathcal{O}(n\,\log^2 k)$	Yes	[4]
AtMost1	Seq. Counter	$\mathcal{O}(n)$	Yes	[102]
	Bitwise	$\mathcal{O}(n \log n)$	Yes	[41, 91]

9.5.4 Optimization

In many settings, the problem to be solved involves a set of constraints ($\mathcal{F}$) subject to a linear cost function $f = \sum_{x \in X} x$. In Boolean domains, optimization problems can be described as follows:

$$\begin{array}{ll} \min & \sum_{i=1}^{n} c_j\, x_j \\ \text{s.t.} & \mathcal{F} \end{array} \tag{3}$$

(3) can be solved with algorithms for pseudo-Boolean optimization. For this concrete case, the cost function can be optimized with standard linear or binary search (see, e.g., [95] for an overview).

Alternatively, (3) can be reduced to weighted partial Maximum Satisfiability (e.g., [51]). The original constraints $\mathcal{F}$ are set as hard clauses. Moreover, each term in the cost function can be represented as a soft clause $(\neg x_j)$ with cost c_j. A wealth of algorithms have been developed in recent years for MaxSAT. These include branch-and-bound search, iterative SAT solving and (unsatisfiable) core-guided approaches. Recent accounts are provided in [3, 66, 85].

9.5.5 Model Enumeration

In many settings, a SAT solver is required to compute all satisfying assignments. A well-known example is in model checking [59, 83]. Another well-known example is the use of SAT solvers in lazy SMT solvers [12], where satisfying assignments are iteratively computed until a model of the SMT formula is found, or the formula is proved unsatisfiable. An essential step in model enumeration is the identification of prime implicants, e.g., [93].

Given a (total) satisfying assignment for the variables, a prime implicant can be obtained by iteratively checking whether each variable is required for satisfying the formula [93]. The resulting set of literals is a prime implicant, and its complement can be used for blocking the recomputation of any model that is covered by the prime implicant.

9.5.6 Minimal Sets

A number of applications of SAT solvers involve computing minimal sets. Concrete examples include computing minimal unsatisfiable subsets (MUSes) [16, 46], minimal correction subsets (MCSes) [67, 75], prime implicates (PIs) [23], and minimal models [17, 18], among many others. Recent work shows that all these problems can be solved with the same algorithms [76]. Algorithms for computing minimal sets of Boolean formulas include the following:

- Insertion-based (or constructive) [104].
- Deletion-based (or destructive) [10, 28].
- Dichotomic [50].
- QuickXplain [60].
- Progression [76].

Of these, QuickXplain and Progression offer the best performance in terms of the worst case number of calls to a SAT solver. The deletion-based algorithm is well known, and has been rediscovered in different settings, e.g. [10, 28]. Given a reference set of elements and a monotone predicate P, each element is iteratively removed from the reference set and the predicate is checked on the resulting set. If the predicate holds, the element is dropped from the reference set; otherwise it is kept. In the end, the resulting set is a minimal set.

Depending on the type of minimal set being computed, different approaches exist for reducing the number of calls to a SAT solver. For computing MUSes and PIs, existing techniques include using unsatisfiable cores to remove unnecessary clauses [10, 16, 35] and model rotation [16, 114].

9.5.7 Quantification

Quantified Boolean Formulas (QBF) are Boolean formulas where the variables can either be existentially or universally quantified. Quantification changes the complexity class, and QBF is a well-known PSPACE-Complete problem [26, 62]. In practice, solving QBF formulas turns out to be significantly harder than solving SAT. A large number of approaches have been proposed for deciding QBF formulas, i.e., for deciding whether a formula is true or false. A recent overview of algorithms for QBF is provided in [43, 56].

9.6 Research Directions

Despite a well-defined set of key techniques, CDCL SAT solvers have been the subject of continued improvements over the years. This section outlines possible lines of research in the area of propositional SAT solving.

One recent promising area of research is the integration of extended resolution into SAT solvers [5, 53]. Extended resolution allows definitions to be created (e.g., new variables representing some Boolean expression). This can provide an added degree of flexibility in modern CDCL SAT solvers.

Another recent promising area of research is to use the DPLL(T) paradigm [88] in designing problem-specific SAT-based algorithms. Concrete examples include specific solvers for handling SAT problems with parity constraints [65], and also PBO solvers [71].

One additional area for future improvements to SAT solvers is formula simplification, before or during search [37, 57, 58].

Besides improvements to SAT solver technology, a number of additional research directions can be envisioned in the area of SAT solving. First, applications of SAT continue to be proposed on a regular basis. This is expected to continue in the future. A related topic is the development of improvements to existing applications of SAT. Moreover, the general area of SAT-based problem solving has been the subject of remarkable improvements in recent years, namely in terms of the many uses of SAT solvers as oracles for solving function problems. Concrete examples include Maximum Satisfiability (MaxSAT) and Pseudo-Boolean Optimization (PBO) [66, 95], minimal unsatisfiable subsets (MUSes) [16, 27], minimal correction subsets (MCSes) [75], backbones of Boolean formulas [77, 121], minimal models, and, in general, minimal sets over monotone predicates [76].

One final area of research is Quantified Boolean Formulas (QBF). Despite the many improvements made in recent years, improvements to QBF solvers are still far inferior to those made to SAT solvers. Nevertheless, recent new uses of SAT solvers in QBF solving suggest further improvements are to be expected [56].

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Mayer-Eichberger, V.: A new look at BDDs for pseudo-boolean constraints. J. Artif. Intell. Res. **45**, 443–480 (2012)
2. Achá, R.J.A., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Practical algorithms for unsatisfiability proof and core generation in SAT solvers. AI Commun. **23**(2–3), 145–157 (2010)
3. Ansótegui, C., Bonet, M.L., Levy, J.: SAT-based MaxSAT algorithms. Artif. Intell. **196**, 77–105 (2013)
4. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: a theoretical and empirical study. Constraints **16**(2), 195–221 (2011)
5. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: Fox, M., Poole, D. (eds.) AAAI Conf. on Artificial Intelligence (AAAI), pp. 15–20. AAAI Press, Palo Alto (2010)

6. Audemard, G., Simon, L.: Experimenting with small changes in conflict-driven clause learning algorithms. In: Stuckey, P.J. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 5202, pp. 630–634. Springer, Heidelberg (2008)
7. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 399–404. IJCAI/AAAI Press, Melbourne/Palo Alto (2009)
8. Bailleux, O., Boufkhad, Y., Roussel, O.: New encodings of pseudo-boolean constraints into CNF. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 5584, pp. 181–194. Springer, Heidelberg (2009)
9. Baker, A.B.: The hazards of fancy backtracking. In: Hayes-Roth, B., Korf, R.E. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 288–293. AAAI Press/MIT Press, Palo Alto/Cambridge (1994)
10. Bakker, R.R., Dikker, F., Tempelman, F., Wognum, P.M.: Diagnosing and solving overdetermined constraint satisfaction problems. In: Bajcsy, R. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 276–281. Morgan Kaufmann, Cambridge (1993)
11. Baptista, L., Marques-Silva, J.: Using randomization and learning to solve hard real-world instances of satisfiability. In: Dechter [34], pp. 489–494
12. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
13. Batcher, K.E.: Sorting networks and their applications. In: AFIPS Spring Joint Computer Conference. AFIPS Conference Proceedings, vol. 32, pp. 307–314. Thomson Book Co., Washington D.C. (1968)
14. Bayardo, R.J. Jr., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Kuipers, B., Webber, B.L. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 203–208. AAAI Press/MIT Press, Palo Alto/Cambridge (1997)
15. Belov, A., Chen, H., Mishchenko, A., Marques-Silva, J.: Core minimization in SAT-based abstraction. In: Macii, E. (ed.) Design, Automation & Test in Europe (DATE), pp. 1411–1416. EDA Consortium/ACM, San Jose/New York (2013)
16. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2), 97–116 (2012)
17. Ben-Eliyahu, R., Dechter, R.: On computing minimal models. Ann. Math. Artif. Intell. **18**(1), 3–27 (1996)
18. Ben-Eliyahu-Zohary, R., Palopoli, L.: Reasoning with minimal models: efficient algorithms and applications. Artif. Intell. **96**(2), 421–449 (1997)
19. Biere, A.: PicoSAT essentials. J. Satisf. Boolean Model. Comput. **4**(2–4), 75–97 (2008)
20. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
21. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
22. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
23. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Form. Asp. Comput. **20**(4–5), 379–405 (2008)
24. Brafman, R.I.: A simplifier for propositional formulas with many binary clauses. IEEE Trans. Syst. Man Cybern., Part B, Cybern. **34**(1), 52–59 (2004)
25. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Trans. Comput. **35**(8), 677–691 (1986)
26. Büning, H.K., Bubeck, U.: Theory of quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 735–760. IOS Press, Amsterdam (2009)

27. Büning, H.K., Kullmann, O.: Minimal unsatisfiability and autarkies. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 339–401. IOS Press, Amsterdam (2009)
28. Chinneck, J.W., Dravnieks, E.W.: Locating minimal infeasible constraint sets in linear programs. INFORMS J. Comput. **3**(2), 157–168 (1991)
29. Coelho, H., Studer, R., Wooldridge, M. (eds.): Proceedings of the ECAI 2010—19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16–20, 2010. IOS Press, Amsterdam (2010)
30. Cook, S.A.: The complexity of theorem-proving procedures. In: Harrison, M.A., Banerji, R.B., Ullman, J.D. (eds.) Annual Symp. on Theory of Computing (STOC), pp. 151–158. ACM, New York (1971)
31. Crawford, J.M., Auton, L.D.: Experimental results on the crossover point in satisfiability problems. In: Fikes, R., Lehnert, W.G. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 21–27. AAAI Press/MIT Press, Palo Alto/Cambridge (1993)
32. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
33. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
34. Dechter, R. (ed.): Principles and Practice of Constraint Programming—CP 2000, Proceedings of the 6th International Conference, Singapore, September 18–21, 2000. LNCS, vol. 1894. Springer, Heidelberg (2000)
35. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: Biere, A., Gomes, C.P. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
36. Dubois, O., André, P., Boufkhad, Y., Carlier, J.: SAT versus UNSAT. In: Johnson, D.S., Trick, M. (eds.) Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, pp. 415–436. AMS, Providence (1996)
37. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
38. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2003)
39. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. J. Satisf. Boolean Model. Comput. **2**(1–4), 1–26 (2006)
40. Freeman, J.W.: Improvements to propositional satisfiability search algorithms. Ph.D. thesis, University of Pennsylvania (1995)
41. Frisch, A.M., Peugniez, T.J.: Solving non-boolean satisfiability problems with stochastic local search. In: Nebel, B. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 282–290. Morgan Kaufmann, Cambridge (2001)
42. Fujiwara, H., Shimono, T.: On the acceleration of test generation algorithms. Trans. Comput. **32**(12), 1137–1144 (1983)
43. Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 761–780. IOS Press, Amsterdam (2009)
44. Goldberg, E.I., Novikov, Y.: BerkMin: a fast and robust Sat-solver. In: Design, Automation & Test in Europe (DATE), pp. 142–149. IEEE, Piscataway (2002)
45. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: Mostow, J., Rich, C. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 431–437. AAAI Press/MIT Press, Palo Alto/Cambridge (1998)

46. Grégoire, É., Mazure, B., Piette, C.: On approaches to explaining infeasibility of sets of boolean clauses. In: Intl. Conf. on Tools with Artificial Intelligence (ICTAI), vol. 1, pp. 74–83. IEEE, Piscataway (2008)
47. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and intensification in parallel SAT solving. In: Cohen, D. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 6308, pp. 252–265. Springer, Heidelberg (2010)
48. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. J. Satisf. Boolean Model. Comput. **6**(4), 245–262 (2009)
49. Hamadi, Y., Wintersteiger, C.M.: Seven challenges in parallel SAT solving. AI Mag. **34**(2), 99–106 (2013)
50. Hemery, F., Lecoutre, C., Sais, L., Boussemart, F.: Extracting MUCs from constraint networks. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) European Conf. on Artificial Intelligence (ECAI), pp. 113–117. IOS Press, Amsterdam (2006)
51. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSAT: an efficient weighted Max-SAT solver. J. Artif. Intell. Res. **31**, 1–32 (2008)
52. Huang, J.: The effect of restarts on the efficiency of clause learning. In: Veloso, M.M. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 2318–2323 (2007)
53. Huang, J.: Extended clause learning. Artif. Intell. **174**(15), 1277–1284 (2010)
54. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 14–25. ACM, New York (2000)
55. Janhunen, T., Niemelä, I.: Compact translations of non-disjunctive answer set programs to propositional clauses. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning—Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday. Lecture Notes in Artificial Intelligence, vol. 6565, pp. 111–130. Springer, Heidelberg (2011)
56. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012)
57. Järvisalo, M., Biere, A., Heule, M.: Simulating circuit-level simplifications on CNF. J. Autom. Reason. **49**(4), 583–619 (2012)
58. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)
59. Jin, H., Han, H., Somenzi, F.: Efficient conflict analysis for finding all satisfying assignments of a boolean circuit. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 287–300. Springer, Heidelberg (2005)
60. Junker, U.: QuickXplain: preferred explanations and relaxations for over-constrained problems. In: McGuinness, D.L., Ferguson, G. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 167–172. AAAI Press/MIT Press, Palo Alto/Cambridge (2004)
61. Kautz, H.A., Selman, B.: Planning as satisfiability. In: Neumann, B. (ed.) European Conf. on Artificial Intelligence (ECAI), pp. 359–363. Wiley, New York (1992)
62. Kleine-Büning, H., Letterman, T.: Propositional Logic: Deduction and Algorithms. Cambridge University Press, Cambridge (1999)
63. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Heidelberg (2008)
64. Kullmann, O. (ed.): Theory and Applications of Satisfiability Testing—SAT 2009, Proceedings of the 12th International Conference, SAT 2009, Swansea, UK, June 30–July 3, 2009. LNCS, vol. 5584. Springer, Heidelberg (2009)
65. Laitinen, T., Junttila, T.A., Niemelä, I.: Extending clause learning DPLL with parity reasoning. In: Coelho et al. [29], pp. 21–26
66. Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 613–631. IOS Press, Amsterdam (2009)

67. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008)
68. Lynce, I., Baptista, L., Marques-Silva, J.: Towards provably complete stochastic search algorithms for satisfiability. In: Brazdil, P., Jorge, A. (eds.) Portuguese Conf. on Artificial Intelligence (EPIA). LNCS, vol. 2258, pp. 363–370. Springer, Heidelberg (2001)
69. Lynce, I., Marques-Silva, J.: Probing-based preprocessing techniques for propositional satisfiability. In: Intl. Conf. on Tools with Artificial Intelligence (ICTAI), pp. 105–110. IEEE, Piscataway (2003)
70. Lynce, I., Marques-Silva, J.: Efficient data structures for backtrack search SAT solvers. Ann. Math. Artif. Intell. **43**(1), 137–152 (2005)
71. Manquinho, V.M., Marques-Silva, J.: Satisfiability-based algorithms for boolean optimization. Ann. Math. Artif. Intell. **40**(3–4), 353–372 (2004)
72. Marques-Silva, J.: Search algorithms for satisfiability problems in combinational switching circuits. Ph.D. thesis, University of Michigan (1995)
73. Marques-Silva, J.: The impact of branching heuristics in propositional satisfiability algorithms. In: Barahona, P., Alferes, J.J. (eds.) Portuguese Conf. on Artificial Intelligence (EPIA). LNCS, vol. 1695, pp. 62–74. Springer, Heidelberg (1999)
74. Marques-Silva, J.: Algebraic simplification techniques for propositional satisfiability. In: Dechter [34], pp. 537–542
75. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: Rossi, F. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 615–622. IJCAI/AAAI, Melbourne/Palo Alto (2013)
76. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in boolean formulae. In: Sharygina, N., Veith, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 592–607. Springer, Heidelberg (2013)
77. Marques-Silva, J., Janota, M., Lynce, I.: On computing backbones of propositional theories. In: Coelho et al. [29], pp. 15–20
78. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press, Amsterdam (2009)
79. Marques-Silva, J., Sakallah, K.A.: Dynamic search-space pruning techniques in path sensitization. In: Lorenzetti, M.J. (ed.) Design Automation Conf. (DAC), pp. 705–711. ACM, New York (1994)
80. Marques-Silva, J., Sakallah, K.A.: GRASP—a new search algorithm for satisfiability. In: Rutenbar, R.A., Otten, R.H.J.M. (eds.) Intl. Conf. on Computer-Aided Design (ICCAD), pp. 220–227. IEEE/ACM, Piscataway/New York (1996)
81. Marques-Silva, J., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. Trans. Comput. **48**(5), 506–521 (1999)
82. Marques-Silva, J., Sakallah, K.A. (eds.): Theory and Applications of Satisfiability Testing—SAT 2007, Proceedings of the 10th International Conference, Lisbon, Portugal, May 28–31, 2007. LNCS, vol. 4501. Springer, Heidelberg (2007)
83. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
84. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
85. Morgado, A., Heras, F., Liffiton, M.H., Planes, J., Marques-Silva, J.: Iterative and core-guided MaxSAT solving: a survey and assessment. Constraints **18**(4), 478–534 (2013)
86. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conf. (DAC), pp. 530–535. ACM, New York (2001)
87. Nam, G.J., Sakallah, K.A., Rutenbar, R.A.: Satisfiability-based detailed FPGA routing. In: Intl. Conf. on VLSI Design, pp. 574–577. IEEE, Piscataway (1999)

88. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
89. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva and Sakallah [82], pp. 294–299
90. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2**(3), 293–304 (1986)
91. Prestwich, S.D.: Variable dependency in local search: prevention is better than cure. In: Marques-Silva and Sakallah [82], pp. 107–120
92. Prestwich, S.D.: CNF encodings. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 75–97. IOS Press, Amsterdam (2009)
93. Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 31–45. Springer, Heidelberg (2004)
94. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
95. Roussel, O., Manquinho, V.M.: Pseudo-boolean and cardinality constraints. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 695–733. IOS Press, Amsterdam (2009)
96. Sabharwal, A., Samulowitz, H., Sellmann, M.: Learning back-clauses in SAT. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 498–499. Springer, Heidelberg (2012)
97. Schulz, M.H., Trischler, E., Sarfert, T.M.: SOCRATES: a highly efficient automatic test pattern generation system. Trans. Comput.-Aided Des. Integr. Circuits Syst. **7**(1), 126–137 (1988)
98. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: Johnson, D.S., Trick, M. (eds.) Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, pp. 521–532. AMS, Providence (1996)
99. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
100. Sheeran, M., Stålmarck, G.: A tutorial on Stålmarck's proof procedure for propositional logic. Form. Methods Syst. Des. **16**(1), 23–58 (2000)
101. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 94–105. IEEE, Piscataway (2003)
102. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: van Beek, P. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)
103. Sinz, C., Iser, M.: Problem-sensitive restart heuristics for the DPLL procedure. In: Kullmann [64], pp. 356–362
104. de Siqueira, J.L.N., Puget, J.F.: Explanation-based generalisation of failures. In: Kodratoff, Y. (ed.) European Conf. on Artificial Intelligence (ECAI), pp. 339–344. Pitman, London (1988)
105. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann [64], pp. 237–243
106. Stallman, R.M., Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artif. Intell. **9**(2), 135–196 (1977)
107. Stephan, P.R., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Combinational test generation using satisfiability. Trans. Comput.-Aided Des. Integr. Circuits Syst. **15**(9), 1167–1176 (1996)

108. Stuckey, P.J.: There are no CNF problems. In: Järvisalo, M., Gelder, A.V. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7962, pp. 19–21. Springer, Heidelberg (2013)
109. Tarjan, R.E.: Finding dominators in directed graphs. SIAM J. Comput. **3**(1), 62–89 (1974)
110. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Silenko, A.O. (ed.) Studies in Constructive Mathematics and Mathematical Logic, Part II, pp. 115–125. Springer, Heidelberg (1968)
111. Urquhart, A.: The complexity of propositional proofs. Bull. Symb. Log. **1**(4), 425–467 (1995)
112. Van Gelder, A., Tsuji, Y.K.: Satisfiability testing with more reasoning and less guessing. Tech. rep., University of California at Santa Cruz (1995)
113. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. Inf. Process. Lett. **68**(2), 63–69 (1998)
114. Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: Milano, M. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 7514, pp. 672–687. Springer, Heidelberg (2012)
115. Wolfram, D.A.: Forward checking and intelligent backtracking. Inf. Process. Lett. **32**(2), 85–87 (1989)
116. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008)
117. Zabih, R., McAllester, D.A.: A rearrangement search strategy for determining propositional satisfiability. In: Shrobe, H.E., Mitchell, T.M., Smith, R.G. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 155–160. AAAI Press/MIT Press, Palo Alto/Cambridge (1988)
118. Zhang, H.: SATO: an efficient propositional prover. In: McCune, W. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 1249, pp. 272–275. Springer, Heidelberg (1997)
119. Zhang, H., Stickel, M.E.: Implementing the Davis-Putnam method. J. Autom. Reason. **24**(1/2), 277–296 (2000)
120. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications. In: Design, Automation & Test in Europe (DATE), pp. 10,880–10,885. IEEE, Piscataway (2003)
121. Zhu, C.S., Weissenbacher, G., Sethi, D., Malik, S.: SAT-based techniques for determining backbones for post-silicon fault localisation. In: Zilic, Z., Shukla, S.K. (eds.) Intl. High Level Design Validation and Test Workshop (HLDVT), pp. 84–91. IEEE, Piscataway (2011)