Handbook of Satisfiability 133
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
IOS Press, 2021
© 2021 Joao Marques-Silva, Ines Lynce, Sharad Malik and IOS Press. All rights reserved.
doi:10.3233/FAIA200987

Chapter 4

Conflict-Driven Clause Learning SAT Solvers

Joao Marques-Silva, Ines Lynce, and Sharad Malik

4.1. Introduction

The decision problem for propositional logic, commonly referred to as the Boolean Satisfiability (SAT) problem, is well-known to be NP-complete [Coo71].[1] Although NP-completeness implies (under the reasonable assumption that P $\neq$ NP) that any complete SAT algorithm runs in worst-case exponential time, it is also the case that SAT solvers most often defy this predicted worst-case behavior. Indeed, SAT can be viewed as a success story of computer science. From an academic curiosity in the early 90s, when solving a formula with a few hundred variables would be considered a significant success, SAT solvers have metamorphosed into highly efficient engines that routinely solve formulas with millions of variables, and which find widespread industrial deployment. Conflict-Driven Clause Learning (CDCL) SAT solving is the sole reason for this success.

The development of CDCL SAT solvers was from the outset motivated by practical applications. During the late 80s and early 90s, applications of SAT in automatic test pattern-generation (ATPG) [Lar89, Lar92] and timing analysis [MSS$^+$91, SBSV96] motivated the development of clause learning and the GRASP SAT algorithm [MS95, MSS96c, MSS99]. Another development was the application of SAT to automated planning [KS92, KS96], but here the initial focus was on exploiting (incomplete) local-search SAT solvers, which reflected the strong prominence of incomplete SAT throughout the 90s [SLM92]. The initial practical success of GRASP [MSS96a] led to the inclusion of GRASP-like clause learning in SATO [Zha97], which in turn was featured in the original SAT-based model checking papers [BCCZ99, BCC$^+$99]. The immediate impact of SAT-based model checking, and the performance improvements achieved with Chaff [MMZ$^+$01, ZMMM01], resulted in a stream of improvements in SAT solvers, and later in SMT solvers [BSST09], which catalyzed the many applications we see today.

[1]Besides the work of S. Cook, NP-completeness was investigated independently by L. Levin [Lev73].

The range of practical applications of CDCL SAT solving has continued to expand in recent years, arguably at an ever increasing pace. The practical significance of CDCL SAT solvers has also influenced the organization of a number of other automated reasoning engines, ranging from the propositional domain to the first-order logic setting, and also finding applications for reasoning about logics more expressive than first-order [GK03, Kor08, KV13, Vor14]. Moreover, CDCL SAT solvers have impacted research in several other areas. The applications of CDCL SAT solvers motivated the development of new propositional encodings [Wal00, Gen02, BB03, GN04, AM04, Sin05, ES06, Pre07, Pre09, RM09, BBR09, ANORC09, CZI10, ANORC11b, ANORC11a, ANO^{+}12, OLH^{+}13, Kar18, MJML14, MPS14]. The usage of CDCL SAT solvers as oracles (for the class NP) motivated extensive revisions of the algorithms for a number of problems, including maximum satisfiability [MSP07, MMSP09, ABL09, DB11, ABL13, IMM^{+}14, MJML14, MDMS14, MHL^{+}13, SBJ16], tackling enumeration problems [NBE12, JMSSS14, PIMMS15, LPMMS16, PMJaMS18, GIL18], solving various quantification problems [JMS11, JKMSC12, RT15, JKMSC16, RS16, IPMS15, IJMS16, SWJ16, IMMS16, CLLB^{+}17, LLBdLM18], and finding minimal sets [GMP08, MSJB13, MS10, Nad10, MSL11, BLMS12, MSHJ^{+}13, NRS13, GLM14a, BDTK14, BK15, MPMS15, NBMS18], among others. Furthermore, CDCL SAT solvers have also led to a series of new results in propositional proof complexity, but also to the search for the automation of proof systems stronger than resolution [BKS04, PD09, AFT09, Hua10, AKS10, PD11, AFT11, IMMS17b, BBI^{+}18, EN18, EGCNV18]. Given the above, it is reasonable to argue that CDCL SAT solving has become one of the mainstays of research in the general areas of constraint solving and optimization, computational logic and automated reasoning.

The purpose of this chapter is to provide an overview of the organization and practical uses of CDCL SAT solvers, but also to give an historical perspective to the key ideas that integrate CDCL SAT solvers. The chapter covers the organization of CDCL SAT solvers as instantiated in most modern tools.[2] Promising techniques, that have been integrated in a smaller number of solvers, will also be described. The approach taken is to describe the key ideas behind CDCL SAT solvers, providing a comprehensive range of examples that illustrate how these ideas work in practice. The chapter also provides a brief overview of the many uses of CDCL SAT solvers, some of which represent existing lines of research. For somewhat different treatments of the same subject, the interested reader is referred to a number of related overviews which have been published over the years [MS98, MSS00, ZM02, LMS02, LMS03, Mit05, CESS08, MS08, Bie08b, GKSS08, MSLM09, MZ09, Knu15, VWM15, MSM18].

The chapter is organized as follows. Section 4.2 introduces the notation used in the remainder of the chapter. Section 4.3 describes the implementation of CDCL SAT solvers. Section 4.4 outlines the ways in which CDCL SAT solvers are used in practical settings. Section 4.5 provides a glimpse of the practical and theoretical significance of CDCL SAT solving. Moreover, Section 4.6 pro-

[2]Compared to the first edition of the Handbook of Satisfiability, the chapter has been extensively rewritten, with the purpose of providing a more modern treatment of CDCL, taking into account relevant work over the last decade, but also aiming to provide a glimpse into the many practical successes of CDCL SAT solving.

vides a brief historical perspective on the development of CDCL SAT solvers. Section 4.7 provides additional pointers for examples of SAT solvers as well as the PySAT framework for prototyping with SAT solvers [IMMS18]. Finally, Section 4.8 summarizes the chapter but also research directions for the improvement and application of CDCL SAT solvers.

4.2. Preliminaries

This section introduces the notation and definitions used throughout. The goal is to describe variants of backtrack search algorithms, and so the definitions reflect this goal. Some notation is adapted from [BL99, BHvMW09].

A set $\mathbb{V}$ of propositional variables is assumed. Variables will be lower case letters, with or without indices, e.g. $\{a, b, c, \ldots, x_1, x_2, \ldots, y_1, y_2, \ldots\}$. Propositional formulas are defined inductively over $\mathbb{V}$, using the standard logical connectives $\neg$, $\vee$ and $\wedge$, as follows:

1. $\mathcal{F} = x$, with $x \in \mathbb{V}$, is a propositional formula.
2. If $\mathcal{F}$ is a propositional formula, then $(\neg\mathcal{F})$ is a propositional formula.
3. If $\mathcal{F}$ and $\mathcal{G}$ are propositional formulas, then $(\mathcal{F} \vee \mathcal{G})$ is a propositional formula.
4. If $\mathcal{F}$ and $\mathcal{G}$ are propositional formulas, then $(\mathcal{F} \wedge \mathcal{G})$ is a propositional formula.

The definition can be extended to consider other logical connectives.

Most SAT solvers operate on the more restricted representation of Conjunctive Normal Form (CNF) formulas. Each clause $\mathfrak{c}_i$ is defined as a set of literals. A literal is either a variable $x \in \mathbb{V}$ or its complement $\bar{x}$. [3] We also use $var(l_i)$ to denote the variable associated with some literal l_i. A set of clauses will be interpreted as a conjunction of clauses. A clause will be interpreted as a disjunction of literals. Throughout this chapter, both representations will be used.

Example 4.2.1 (CNF Formula)**.** An example of a CNF formula is:

$$\mathcal{F} = \{\{a, \bar{b}\}, \{b, c\}, \{b, \bar{d}\}, \{\bar{a}, \bar{c}, d\}\} \tag{4.1}$$

The alternative clausal representation is:

$$\mathcal{F} = (a \vee \bar{b}) \wedge (b \vee c) \wedge (b \vee \bar{d}) \wedge (\bar{a} \vee \bar{c} \vee d) \tag{4.2}$$

Propositional variables can be assigned a propositional value, taken from $\{0, 1\}$. However, in the context of search algorithms for SAT, variables may also be *unassigned*. As a result, assignments to propositional variables are defined as a function $\nu : \mathbb{V} \to \{0, \mathfrak{u}, 1\}$, where $\mathfrak{u}$ denotes an *undefined* value used when a variable has not been assigned a value in $\{0, 1\}$. (ν could also be defined as a partial map from $\mathbb{V}$ to $\{0, 1\}$, but we opt to use the additional value $\mathfrak{u}$.) Given an assignment ν, if all variables are assigned a value in $\{0, 1\}$, then ν is referred to as a *complete assignment*. Otherwise it is a *partial assignment*.

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

[3] To keep the notation as simple as possible, we will represent negative literals by $\bar{x}$ instead of the often used notation $\neg x$.

1. If $\mathcal{F} = x$, with $x \in \mathbb{V}$, then $\mathcal{F}^\nu = \nu(x)$.
2. If $\mathcal{F} = (\neg \mathcal{G})$, then
$$\mathcal{F}^\nu = \begin{cases} 0 & \text{if } \mathcal{G}^\nu = 1 \\ 1 & \text{if } \mathcal{G}^\nu = 0 \\ \mathfrak{u} & \text{otherwise} \end{cases}$$
3. If $\mathcal{F} = (\mathcal{E} \vee \mathcal{G})$, then
$$\mathcal{F}^\nu = \begin{cases} 1 & \text{if } \mathcal{E}^\nu = 1 \text{ or } \mathcal{G}^\nu = 1 \\ 0 & \text{if } \mathcal{E}^\nu = 0 \text{ and } \mathcal{G}^\nu = 0 \\ \mathfrak{u} & \text{otherwise} \end{cases}$$
4. If $\mathcal{F} = (\mathcal{E} \wedge \mathcal{G})$, then
$$\mathcal{F}^\nu = \begin{cases} 1 & \text{if } \mathcal{E}^\nu = 1 \text{ and } \mathcal{G}^\nu = 1 \\ 0 & \text{if } \mathcal{E}^\nu = 0 \text{ or } \mathcal{G}^\nu = 0 \\ \mathfrak{u} & \text{otherwise} \end{cases}$$

The definition of value of a formula applies to arbitrary propositional formulas, and so also to CNF formulas. An assignment that satisfies a propositional formula is referred to as a *model* (or as a *satisfying assignment*).

The assignment function ν will also be viewed as a set of tuples (x_i, v_i), with $v_i \in \{0, 1\}$. Adding a tuple (x_i, v_i) to ν corresponds to assigning v_i to x_i, such that $\nu(x_i) = v_i$. Removing a tuple (x_i, v_i) from ν, with $\nu(x_i) \neq u$, corresponds to assigning u to x_i. With a mild abuse of notation, we use $x_i = v_i$ when $\nu(x_i) = v_i$ and ν is clear from the context. Moreover, variable assignments will also be represented as (true) literals. Concretely, if x_i is used, then it indicates that $\nu(x_i) = 1$, whereas if $\overline{x_i}$ is used, then it indicates that $\nu(x_i) = 0$.

Clauses are characterized as *falsified*, *satisfied*, *unit* or *unresolved.* A clause is falsified if all its literals are assigned value 0. A clause is satisfied if at least one of its literals is assigned value 1. A clause is unit if all literals but one are assigned value 0, and the remaining literal is unassigned. Finally, a clause is unresolved if it is neither unsatisfied, nor satisfied, nor unit.

A key procedure in SAT solvers is the *unit clause rule* [DP60]: 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) [ZM88]. In modern CDCL solvers, as in most implementations of the DPLL algorithm [DP60, DLL62], logical consequences are derived with unit propagation. Unit propagation is applied after each branching step but also during preprocessing,[4] and is used for identifying variables which must be assigned a specific Boolean value. If an unsatisfied clause is identified, a *conflict* condition is declared, and the algorithm needs to backtrack, and may implement some sort of conflict analysis procedure.

Example 4.2.2. Consider the following formula:

$$\begin{aligned} \mathcal{F} &= \mathfrak{c}_1 \wedge \mathfrak{c}_2 \wedge \mathfrak{c}_3 \wedge \mathfrak{c}_4 \\ &= (a) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d) \end{aligned}$$

[4]A branching step is a choice of a variable and a propositional value to assign to the variable.

The operation of unit propagation in this example works as follows. Clause $\mathfrak{c}_1$ is unit. As a result, a must be assigned value 1. Moreover, since $a = 1$, then the clauses $\mathfrak{c}_2$ and $\mathfrak{c}_3$ are also unit (since the literal $\bar{a} = 0$). Thus, the assignments $b = 1$ and $c = 1$ are necessary. Finally, because of these two assignments, then clause $\mathfrak{c}_4$ is also unit, and so the assignment $d = 1$ is also necessary.

In CDCL SAT solvers, each variable x_i is characterized by a number of properties, including the *value*, the *antecedent* (or *reason*) and the *decision level*, denoted respectively by $\nu(v_i) \in \{0, \mathfrak{u}, 1\}$ (introduced above), $\alpha(x_i) \in \mathcal{F} \cup \{\mathfrak{n}, \mathfrak{d}\}$ (where $\mathfrak{n}, \mathfrak{d}$ are special symbols introduced below), and $\delta(x_i) \in \{\mathfrak{u}, 0, 1, \ldots, |X|\}$. A variable x_i that is assigned a value as the result of applying the unit clause rule is said to be *implied.* The unit clause $\mathfrak{c}_j$ used for implying variable x_i is said to be the antecedent of x_i, $\alpha(x_i) = \mathfrak{c}_j$. For variables that are decision variables the antecedent is $\mathfrak{d}$ and the decision level is the depth of the search tree associated with that decision variable. For unassigned variables, the antecedent is set to $\mathfrak{n}$ and the decision level is set to $\mathfrak{u}$. Hence, antecedents are only assigned a clause identifier for variables whose value is implied by other assignments. Literals inherit decision level and antecedent information from the corresponding variable. For assigned variables, the value of a literal x_i is given by the value of variable x_i and the value of a literal $\overline{x_i}$ is given by $1 - \nu(x_i)$. For unassigned variables, the value of any literal is given by $\nu(x_i) = \nu(\overline{x_i}) = \mathfrak{u}$.

The decision level of a literal l_i, $\delta(l_i)$ is defined as the decision level of corresponding variable x_i, $\delta(x_i)$, if $l_i = x_i$ or $l_i = \neg x_i$. The decision level of a variable x_i (either if implied or if it is a decision variable) 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_i is $\mathfrak{u}$. Moreover, $\delta(x_i) = \mathfrak{u}$ when $\alpha(x_i) = \mathfrak{n}$. 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.* Hence, a variable x_i associated with a decision assignment is characterized by having $\alpha(x_i) = \mathfrak{d}$, $\delta(x_i) > 0$, and $v(x_i) \neq \mathfrak{u}$. For a variable that is not a decision assignment, the decision level is defined as follows:

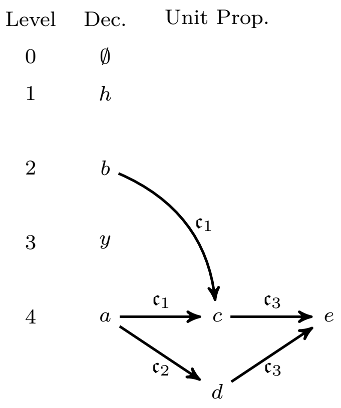
$$\delta(l_i) = \begin{cases} \mathfrak{u} & \text{if } \alpha(l_i) = \mathfrak{n} \\ \max\left(\{0\} \cup \{\delta(l_j) \mid l_j \in \mathit{lits}(\alpha(l_i)) \setminus \{l_i\}\}\right) & \text{otherwise} \end{cases} \tag{4.3}$$

i.e. the decision level of an implied literal is either the highest decision level of the implied literals in a non-unit clause, or it is 0 if the antecedent is unit, i.e. there are no literals in $\alpha(l_i)$ other than l_i. The notation $x_i = v \,@\, d$ is used to denote that $\nu(x_i) = v$ and $\delta(x_i) = d$. Finally, the notation $@\, d$ may be used just to identify the decision level of some anonymous literal.

Example 4.2.3 (Decision Levels & Antecedents)**.** Consider the CNF formula:

$$\begin{aligned} \mathcal{F} &= \mathfrak{c}_1 \wedge \mathfrak{c}_2 \wedge \mathfrak{c}_3 \\ &= (x_1 \vee \overline{x_4}) \wedge (x_1 \vee x_3) \wedge (\overline{x_3} \vee x_2 \vee x_4) \end{aligned}$$

Assume that the decision assignment is $x_4 = 0 \,@\, 1$. Unit propagation yields no additional implied assignments. Assume the second decision is $x_1 = 0 \,@\, 2$. Unit propagation yields the implied assignments $x_3 = 1 \,@\, 2$ and $x_2 = 1 \,@\, 2$. Moreover, $\alpha(x_3) = \mathfrak{c}_2$ and $\alpha(x_2) = \mathfrak{c}_3$.



(a) Implication graph

$x \in \mathbb{V} \cup \{\perp\}$	$\nu(\cdot)$	$\delta(\cdot)$	$\alpha(\cdot)$
h	1	1	$\mathfrak{d}$
b	1	2	$\mathfrak{d}$
y	1	3	$\mathfrak{d}$
a	1	4	$\mathfrak{d}$
c	1	4	$\mathfrak{c}_1$
d	1	4	$\mathfrak{c}_2$
e	1	4	$\mathfrak{c}_3$

(b) State of variables

Figure 4.1. Implication graph for example 4.2.4

During the execution of a search-based SAT solver (which will be detailed later in this chapter), assigned variables as well as their antecedents define a directed acyclic graph $I = (V_I, E_I)$, referred to as the *implication graph* [MS95, MSS96c].

The vertices in the implication graph are defined by all assigned variables and one special node $\perp$, $V_I \subseteq \mathbb{V} \cup \{\perp\}$. The edges in the implication graph are obtained from the antecedent of each assigned variable: if $\mathfrak{c}_j = \alpha(x_i)$, then there is a directed edge from each variable in $\mathfrak{c}_j$, other than x_i, to x_i. If unit propagation yields an falsified clause $\mathfrak{c}_j$, then a special vertex $\perp$ is used to represent the unsatisfied clause. In this case, the antecedent of $\perp$ is defined by $\alpha(\perp) = \mathfrak{c}_j$.

Example 4.2.4 (Implication Graph without Conflict)**.** Consider the CNF formula:

$$\begin{aligned}\mathcal{F}_1 &= \mathfrak{c}_1 \wedge \mathfrak{c}_2 \wedge \mathfrak{c}_3 \\ &= (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee d) \wedge (\bar{c} \vee \bar{d} \vee e)\end{aligned} \tag{4.4}$$

Assume the decision assignment $b = 1@2$. Moreover, assume that the current decision assignment is $a = 1@4$. The resulting implication graph is shown in figure 4.1.[5]

Example 4.2.5 (Implication Graph with Conflict)**.** Consider the CNF formula:

$$\begin{aligned}\mathcal{F}_2 &= \mathfrak{c}_1 \wedge \mathfrak{c}_2 \wedge \mathfrak{c}_3 \wedge \mathfrak{c}_4 \wedge \mathfrak{c}_5 \wedge \mathfrak{c}_6 \\ &= (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee d) \wedge (\bar{c} \vee \bar{d} \vee e) \wedge (\bar{h} \vee \bar{e} \vee f) \wedge (\bar{e} \vee g) \wedge (\bar{f} \vee \bar{g})\end{aligned} \tag{4.5}$$

Assume decision assignments $h = 1@1$, $b = 1@2$ and $y = 1@3$. Moreover, assume that the current decision assignment is $a = 0@4$. The resulting implication graph, that takes into account unit propagation at decision level 4, is shown in figure 4.2, and yields a conflict because clause $(\bar{f} \vee \bar{g})$ becomes falsified.

[5]To keep the notation as simple as possible, all variables are assigned value 1 in the examples, either as decision assignments or as implied assignments. A few exceptions will be noted.

$$\begin{tikzcd}
\text{Level} & \text{Dec.} & \text{Unit Prop.} & & & \\
0 & \emptyset & & & & \\
1 & h \arrow[rrrddd, bend left, "\mathfrak{c}_4"] & & & & \\
2 & b \arrow[rdd, bend left, "\mathfrak{c}_1"] & & & & \\
3 & y & & & & \\
4 & a \arrow[r, "\mathfrak{c}_1"] \arrow[rd, "\mathfrak{c}_2"'] & c \arrow[r, "\mathfrak{c}_3"] & e \arrow[r, "\mathfrak{c}_4"] \arrow[rd, "\mathfrak{c}_5"'] & f \arrow[r, "\mathfrak{c}_6"] & \bot \\
 & & d \arrow[ru, "\mathfrak{c}_3"'] & & g \arrow[ru, "\mathfrak{c}_6"'] &
\end{tikzcd}$$

(a) Implication graph

$x \in \mathbb{V} \cup \{\bot\}$	$\nu(\cdot)$	$\delta(\cdot)$	$\alpha(\cdot)$
h	1	1	$\mathfrak{d}$
b	1	2	$\mathfrak{d}$
y	1	3	$\mathfrak{d}$
a	1	4	$\mathfrak{d}$
c	1	4	$\mathfrak{c}_1$
d	1	4	$\mathfrak{c}_2$
e	1	4	$\mathfrak{c}_3$
f	1	4	$\mathfrak{c}_4$
g	1	4	$\mathfrak{c}_5$
$\bot$	–	–	$\mathfrak{c}_6$

(b) State of variables

Figure 4.2. Implication graph for example 4.2.5

```
Input   : CNF Formula $\mathcal{F}$
Output: Outcome $st$

DPLL ($\mathcal{F}$)
    DLevel ← 0
    if not UnitPropagation() then
        return false
    while not AllVariablesAssigned() do
        DLevel ← DLevel + 1
        (var, val) ← PickBranchVariable()
        Assign(var, val, DLevel, $\mathfrak{d}$)
        while not UnitPropagation() do
            BLevel ← NextUntoggledDecision()
            if BLevel == 0 then
                return false
            Backtrack(BLevel)
            DLevel ← BLevel
            (var, val) ← ToggleDecision(DLevel)
            Assign(var, val, DLevel, $\mathfrak{d}$)
    return true
```

Algorithm 1: The DPLL Algorithm

It should be observed that the notation introduced above is relevant for describing the operation of a modern CDCL SAT solver. A DPLL-like SAT solver usually manipulates significantly less information.

The high-level organization of DPLL [DLL62] is shown in Algorithm 1.

Finally, the original justification for the DPLL algorithm was the propositional resolution operation [DP60]:[6]

$$\frac{x \vee \alpha \quad \bar{x} \vee \beta}{\alpha \vee \beta} \tag{4.6}$$

[6]Robinson's well-known resolution principle for first-order logic was proposed at a later date [Rob65].

Resolution is a complete decision procedure for CNF formulas [DP60], but impractical in practice. The DPLL algorithm represented an attempt at devising a practically effective solution based on resolution [DLL62]. For historical reasons, the acronym DPLL (Davis-Putnam-Logemann-Loveland) reflects the origins of the DPLL algorithm.

4.3. Implementing CDCL SAT Solvers

This section describes the organization of CDCL SAT solvers, starting from the implementation of DPLL outlined in the previous section. The section is broadly organized into two main parts. The first part covers the CDCL techniques that are commonly employed by most modern CDCL SAT solvers. The second part covers other less used techniques.

4.3.1. Organization of a CDCL SAT Solver

The CDCL SAT algorithm shares similarities with the standard backtracking search procedure also used in the DPLL algorithm, where unit propagation is executed after each decision assignment (which creates a new decision level). However, a few fundamental differences exist. First, backtracking can be non-chronological and be related with learning clauses from conflicts [MS95, MSS96c, MSS99]. Second, backtracking occurs after every conflict [MMZ$^+$01, ZMMM01]. Third, the algorithm may decide to *restart* the search from the first decision level [GSC97, GSK98, GSCK00, BMS00]. Furthermore, there are a few additional differences with respect to DPLL, in that learned clauses may eventually be deleted, and highly efficient (lazy) data structures are used. The top-level organization of the CDCL algorithm is shown in Algorithm 2. The following sections detail the differences of CDCL-like algorithms with respect to the DPLL algorithm.

4.3.1.1. Clause Learning

This section provides an overview of one of the hallmarks of CDCL SAT solvers, namely *clause learning.* During backtrack search, clauses are learned after falsified clauses are identified, using a procedure referred to as *conflict analysis* [MS95, MSS96c, MSS96b, MSS96a, MSS99].

4.3.1.1.1. Conflict Analysis. Given a falsified clause, conflict analysis iteratively traces antecedents of variables assigned at the current decision level, and records dependencies of variables assigned at smaller decision levels. The algorithm operates as follows. Variables assigned at the current decision level are analyzed in a first-in first-out fashion (thus being manipulated through a queue), ensuring that a variable is analyzed only after all variables it implies have been analyzed. At each step, the head of the queue is extracted, and its antecedent analyzed. Any literals assigned at decision levels smaller than the current one are added to (i.e. recorded in) the clause being learned. The antecedent tracing procedure terminates when the decision variable for the current decision level is visited. Clearly,

```
Input  : CNF Formula $\mathcal{F}$
Output: Outcome st

CDCL ($\mathcal{F}$)
    DLevel ← 0
    if not UnitPropagation() then
        return false
    while not AllVariablesAssigned() do
        DLevel ← DLevel + 1
        (var, val) ← PickBranchVariable()
        Assign(var, val, DLevel, $\mathfrak{d}$)
        while not UnitPropagation() do
            if DLevel == 0 then
                return false
            BLevel ← ConflictAnalysis()
            Backtrack(BLevel)
            DLevel ← BLevel
            if TimeToRestart() then
                Backtrack(0)
                DLevel ← 0
    return true
```

Algorithm 2: The CDCL Algorithm

Table 4.1. Execution of conflict analysis procedure

Step	Var Queue	Extract Var	Antecedent	Recorded Lits	Added to Queue
0	–	$\perp$	$\mathfrak{c}_6$	$\emptyset$	$\{f, g\}$
1	$[f, g]$	f	$\mathfrak{c}_4$	$\{\bar{h}\}$	$\{e\}$
2	$[g, e]$	g	$\mathfrak{c}_5$	$\{\bar{h}\}$	$\emptyset$
3	$[e]$	e	$\mathfrak{c}_3$	$\{\bar{h}\}$	$\{c, d\}$
4	$[c, d]$	c	$\mathfrak{c}_1$	$\{\bar{h}, \bar{b}\}$	$\{a\}$
4	$[d, a]$	d	$\mathfrak{c}_2$	$\{\bar{h}, \bar{b}\}$	$\emptyset$
5	$[a]$	a	$\mathfrak{d}$	$\{\bar{h}, \bar{b}, \bar{a}\}$	–
6	$[]$	–	–	$\{\bar{h}, \bar{b}, \bar{a}\}$	–

the conflict analysis procedure runs in worst-case linear time in the number of literals.

Example 4.3.1. For the implication graph from Example 4.2.5, the conflict analysis procedure is illustrated in Table 4.1. At step 0, the initialization step, the falsified clause ($\mathfrak{c}_6$) is analyzed and its variables, f and g, both assigned at decision level 4, are added to the queue of variables to visit. At step 1, the top of the queue is (for example) f. The antecedent of f is $\mathfrak{c}_4$. As a result, the literal $\bar{h}$ is recorded, and the variable e is added to the queue. The same process is repeated for steps 2 to 6. As can be observed, all variables assigned at the current decision level are analyzed in order. Any recorded literals are assigned at decision levels smaller than the current one. Hence, in this case, the learned clause is $(\bar{h} \vee \bar{b} \vee \bar{a})$. For this example, the variables and antecedents that are traced are highlighted in Figure 4.3.

Although CDCL SAT solvers implement the conflict analysis procedure as

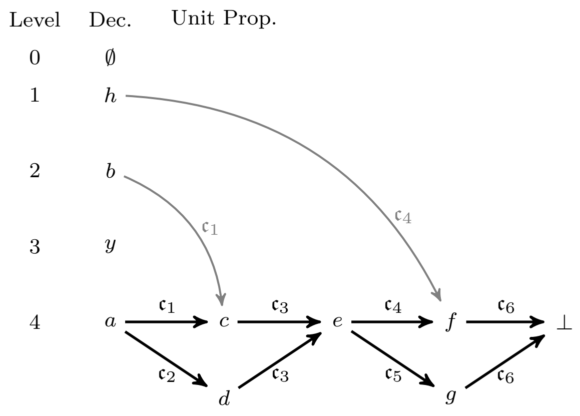


Figure 4.3. Traced antecedents during conflict analysis

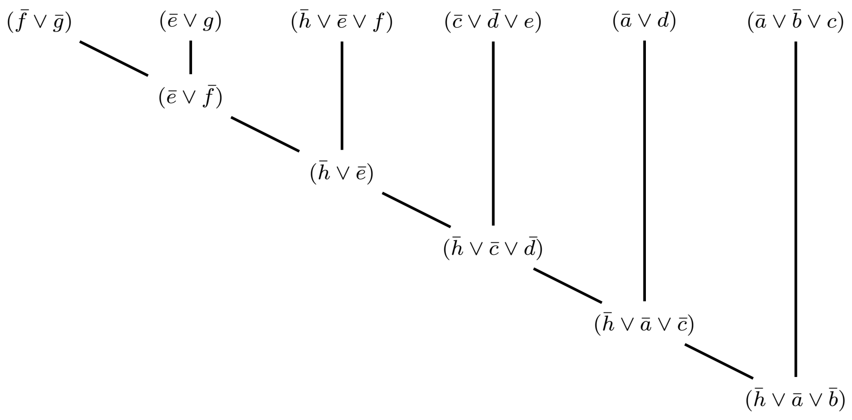

$(\bar{f} \vee \bar{g})$
$(\bar{e} \vee g)$
$(\bar{h} \vee \bar{e} \vee f)$
$(\bar{c} \vee \bar{d} \vee e)$
$(\bar{a} \vee d)$
$(\bar{a} \vee \bar{b} \vee c)$
$(\bar{e} \vee \bar{f})$
$(\bar{h} \vee \bar{e})$
$(\bar{h} \vee \bar{c} \vee \bar{d})$
$(\bar{h} \vee \bar{a} \vee \bar{c})$
$(\bar{h} \vee \bar{a} \vee \bar{b})$

Figure 4.4. Explaining a learned clause with resolution steps

outlined above, there is a tight relationship with resolution, in that learned clauses can be explained with sequences of resolution steps. Concretely, each learned clause obtained from an implication sequence can be explained by *linear input regular resolution* [VG11] (also referred to as trivial resolution steps [BKS04]). As a result, there is a direct mapping between the conflict analysis procedure and linear input regular resolution steps.

Example 4.3.2. Regarding Example 4.3.1, the sequence of resolution steps that explains the learned clause $(\bar{h} \vee \bar{b} \vee \bar{a})$ is shown in Figure 4.4.

A key observation is that although conflict analysis can be explained with resolution steps, to our best knowledge there exists no other approach for automating resolution that is as effective as the conflict analysis procedure devised for CDCL SAT solvers. Even the DPLL algorithm [DLL62, DP60] corresponds to a weaker proof system, i.e. tree-like resolution [BKS04].

Table 4.2. First UIP clause learning

Step	Var Queue	Extract Var	Antecedent	Recorded Lits	Added to Queue
0	–	$\perp$	$\mathfrak{c}_6$	$\emptyset$	$\{f, g\}$
1	$[f, g]$	f	$\mathfrak{c}_4$	$\{\bar{h}\}$	$\{e\}$
2	$[g, e]$	g	$\mathfrak{c}_5$	$\{\bar{h}\}$	$\emptyset$
3	$[e]$	e	$\mathfrak{c}_3$	$\{\bar{h}, \bar{e}\}$	$\emptyset$
6	$[]$	–	–	$\{\bar{h}, \bar{e}\}$	–

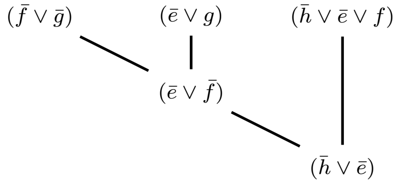
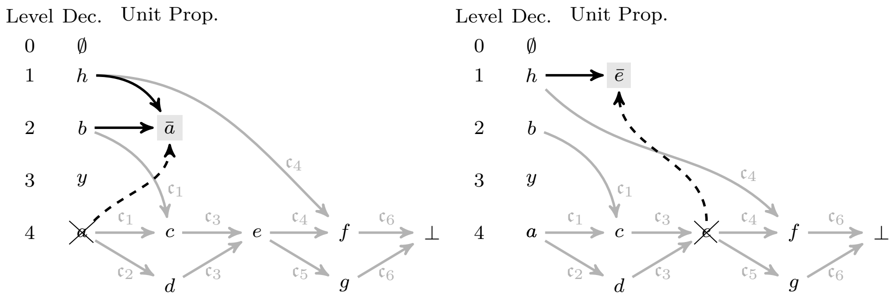


Figure 4.5. Resolution steps with first UIP learning

4.3.1.1.2. Unique Implication Points. The basic clause learning algorithm can be further optimized by taking into account the *structure* of the conflicting implication sequence. As can be observed in the execution of the clause learning procedure in Table 4.1, in step 3 there exists only one variable to trace, concretely e. This variable is referred to as a *unique implication point* (UIP) [MS95, MSS96a, MSS96b, MSS96c, MSS99]. UIPs exhibit a number of important properties. For the implication graph restricted to variables assigned at the current decision level, a UIP is a *dominator* of the decision variable with respect to the conflict node $\perp$. The same assignment to any UIP guarantees that the same conflict will be reproduced. In modern CDCL SAT solvers, clause learning stops at the first UIP [MMZ$^+$01, ZMMM01]. Nevertheless, conflict analysis need not stop at the first UIP, and initial CDCL SAT solvers would learn a clause at each UIP [MS95, MSS96a, MSS99]. This capability of conflict analysis to learn multiple clauses given UIPs is not available in most modern CDCL SAT solvers, despite recent promising results [SSS12].

Example 4.3.3. Regarding Example 4.2.5, with basic conflict analysis shown in Example 4.3.1, e is a UIP. In modern CDCL SAT solvers, the conflict analysis algorithm terminates when this first UIP is reached. In this case, the algorithm learns the clause $(\bar{h} \vee \bar{e})$. The sequence of resolution steps is shown in Figure 4.5. It should be noted that, if conflict analysis were to consider more UIPs, the clause $(\bar{a} \vee \bar{b} \vee e)$ would be learned for UIP a, the decision variable.

4.3.1.1.3. Non-Chronological Backtracking. Learned clauses enable SAT solvers to perform non-chronological backtracking [MS95, MSS96c, MSS99]. A wide range of modern CDCL SAT solvers backtrack (often non-chronologically) after each conflict. This is shown in Algorithm 2. The backtracking decision level is defined as the largest decision level of the literals that remain assigned after conflict analysis. It is immediate to conclude that clause learning at the first UIP



(a) NCB without UIP learning **(b)** NCB with UIP learning

Figure 4.6. Non-chronological backtracking (NCB)

yields the most aggressive backtracking decision level [ABH+08]. Indeed, extending clause learning to any other UIP can only increase the number of traced literals, and so can only maintain or increase the backtracking decision level.

Starting with the SAT solver Chaff, the non-chronological backtracking step would be taken after each conflict [MMZ+01]. This is in contrast with the backtracking strategy used in the GRASP SAT solver[MS95, MSS96c, MSS99], where the non-chronological backtracking step would be taken *only* after the two assignments to a given decision variable had been considered. The backtracking strategy used in GRASP, improved with heuristics to decide when to apply it, has become popular in recent SAT solvers [NR18, MB19], being referred to as a *chronological backtracking* step when implementing *non-chronological backtracking.*

Example 4.3.4. For Example 4.3.1, the learned clause is $(\bar{h} \vee \bar{b} \vee \bar{a})$. After conflict analysis, at decision level 4, the clause becomes unit, due to a becoming unassigned. The largest decision level is 2, and so the CDCL algorithm backtracks to decision level 2, where the learned clause is still unit and implies the assignment $a = 0$. In case conflict analysis stops at the first UIP (see Example 4.3.3), then the learned clause is $(\bar{h} \vee \bar{e})$, the CDCL algorithm backtracks to decision level 1, and the learned clause implies the assignment $e = 0$. As can also be observed from the first UIP clause, exploiting UIPs enables more aggressive backtracking, in this case to decision level 1. Backtracking with and without UIP learning is shown in Figure 4.6.

4.3.1.1.4. Learned Clause Minimization. Although the conflict analysis procedure has remained unchanged since it was first proposed [MS95, MSS96c, MSS99], improvements have been made on how to make the learned clauses more effective in practice, concretely by filtering literals in the learned clause that can be deemed irrelevant. This section overviews two approaches for removing redundant literals from a learned clause [SB09]. One approach exploits the concept of self-subsuming resolution [SP04]. The other approach traces antecedents at decision levels lower than the current one, aiming at discovering subsets of literals in the learned clause that are implied by the other literals.

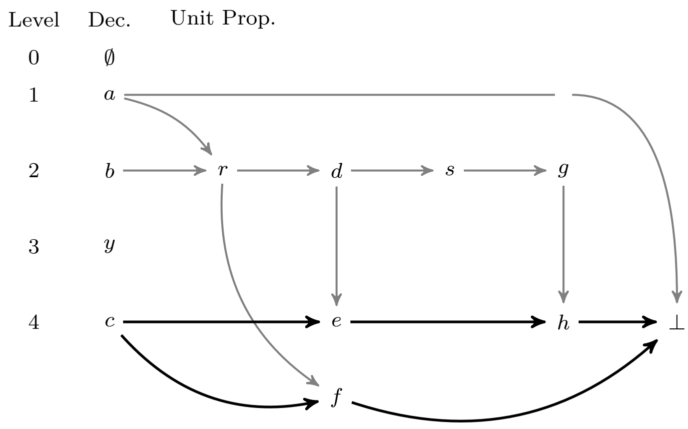


Figure 4.7. Clause minimization example

The first approach for learned clause minimization, referred to as *local minimization*, consists of applying self-subsumption resolution[7] among the literals in the learned clause, aiming at identifying literals that can be dropped from the learned clause.

A more sophisticated (and more time-consuming) procedure, referred to as *recursive minimization*, traverses the implication graph at decision levels *smaller* than the current one:

1. Given a learned clause, pick some candidate literal, and mark all the other literals.
2. Recursively trace antecedents, starting from candidate, until marked literals or decision variables are reached.
3. If only marked literals are visited, then candidate can be declared redundant.
4. Repeat for all other literals in clause (except for decision literals).

Some practical implementations approximate exact recursive minimization [SB09]. Moreover, existing implementations ensure that the amortized run time is linear in the size of the implication graph (and so linear on the number of variables). Additional improvements to conflict clause minimization have been proposed in recent years [VG09, LLX$^+$17].

The relationship of local and recursive learned clause minimization with resolution is immediate. Both clause minimization schemes identify additional resolution steps which enable the learned clause to be simpler.

Example 4.3.5. Consider the following CNF formula:

$$\begin{aligned}\mathcal{F} &= \mathfrak{c}_1 \wedge \mathfrak{c}_2 \wedge \mathfrak{c}_3 \wedge \mathfrak{c}_4 \wedge \mathfrak{c}_5 \wedge \mathfrak{c}_6 \wedge \mathfrak{c}_7 \wedge \mathfrak{c}_8 \\ &= (\bar{a} \vee \bar{b} \vee r) \wedge (\bar{r} \vee d) \wedge (\bar{c} \vee \bar{d} \vee e) \wedge (\bar{c} \vee \bar{d} \vee f) \wedge (\bar{d} \vee s) \wedge \\ &\quad\ (\bar{s} \vee g) \wedge (\bar{e} \vee \bar{g} \vee h) \wedge (\bar{f} \vee \bar{h} \vee \bar{a})\end{aligned}$$

[7]Given $(x \vee \alpha)$ and $(\bar{x} \vee \alpha \vee \beta)$, self-subsuming resolution produces the clause $(\alpha \vee \beta)$.

Table 4.3. Execution of recursive minimization

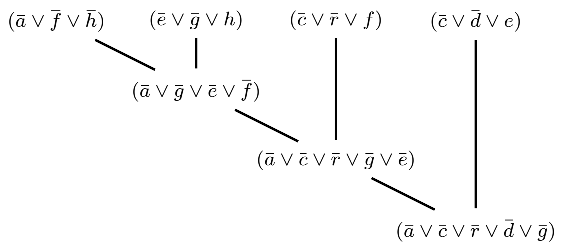
Target	Curr Var	Marked	Unmarked	Vars to Trace	Action
g	g	$\{a, d, r, c\}$	$\emptyset$	$[s]$	–
g	s	$\{a, d, r, c\}$	$\emptyset$	$[d]$	–
g	d	$\{a, d, r, c\}$	$\emptyset$	$[]$	d marked, skip
g	–	$\{a, d, r, c\}$	$\emptyset$	$[]$	no unmarked vars; $\therefore$ drop g
d	d	$\{a, r, c\}$	$\emptyset$	$[r]$	–
d	r	$\{a, r, c\}$	$\emptyset$	$[]$	r marked, skip
d	–	$\{a, r, c\}$	$\emptyset$	$[]$	no unmarked vars; $\therefore$ drop d
r	r	$\{a, c\}$	$\emptyset$	$[a, b]$	–
r	a	$\{a, c\}$	$\emptyset$	$[c]$	a marked
r	b	$\{a, c\}$	$\{b\}$	$[]$	b decision & unmarked
r	–	$\{a, c\}$	$\{b\}$	$[]$	unmarked vars; $\therefore$ keep r
a, c	–	–	$\emptyset$	$[]$	a, c decision variables; keep both

Consider the sequence of decisions: $\langle a = 1, b = 1, y = 1, c = 1\rangle$ and the resulting implication graph, as shown in Figure 4.7. Conflict analysis yields the (first UIP) clause: $(\bar{a} \vee \bar{g} \vee \bar{d} \vee \bar{r} \vee \bar{c})$. Local minimization exploits the fact that self-subsuming resolution between the learned clause and $\mathfrak{c}_2$ reduces the number of literals by one, thus producing: $(\bar{a} \vee \bar{g} \vee \bar{r} \vee \bar{c})$. Recursive minimization enables one further simplification, producing the final minimized clause: $(\bar{a} \vee \bar{r} \vee \bar{c})$. Alternatively, the sole execution of recursive minimization is summarized in Table 4.3. The literal on variable g is dropped because tracing only reaches variable d, which is marked (as it is included in the learned clause). The same applies to d. The literal on variable r cannot be dropped, since tracing reaches b, which is not marked. The literals of variables a and c cannot be dropped, since these are decision variables. Moreover, Figure 4.8 summarizes clause learning and also local and recursive minimization. Figure 4.8a, Figure 4.8b and Figure 4.8c show, respectively, the resolution derivation for the original learned clause, the result with local minimization, and the result with recursive minimization.

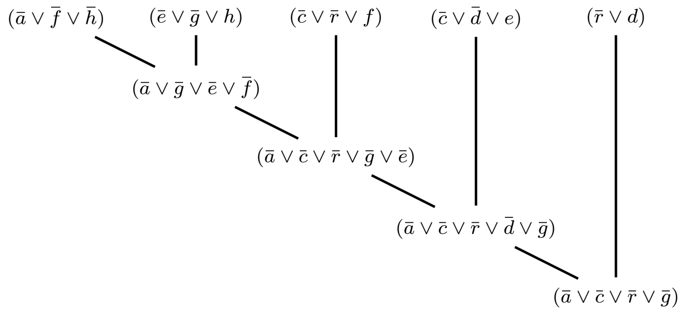
4.3.1.2. Search Restarts

In the late 90s, researchers observed that *branching randomization* would cause DPLL-like SAT solvers to exhibit heavy-tailed behavior on satisfiable formulas [GSC97, GSK98]. Concretely, the run time of a DPLL-like SAT solver on some satisfiable formulas would be small with high probability, but also the probability mass corresponding to large run times would be non-negligible, i.e. the run time distribution on satisfiable instances exhibited a heavy tail [GSC97, GSK98, GSCK00, GS09]. Given this behavior, a natural solution for improving the run time of DPLL-like SAT solvers on satisfiable instances would be to exploit branching randomization and apply a policy to restart the search often, e.g. after a number of conflicts.

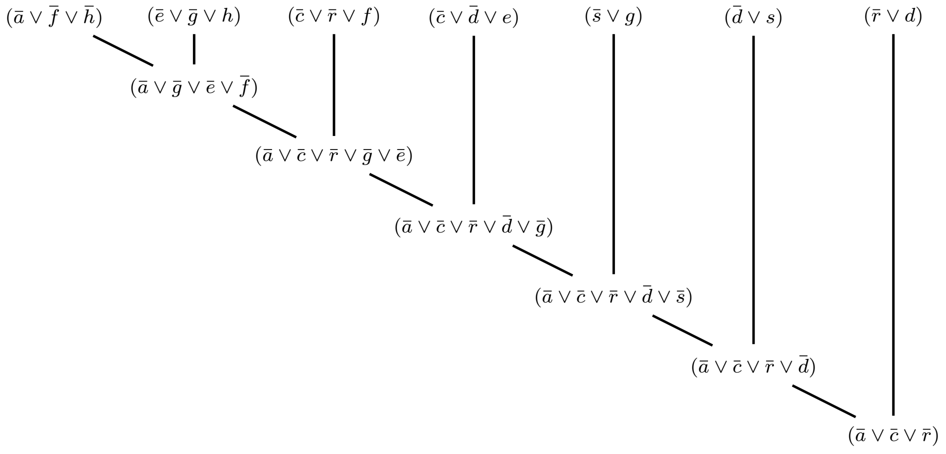
Example 4.3.6 (Search Restarts)**.** The example sketched in Figure 4.9 illustrates the execution of restarts. After a cutoff, the search is restarted. For completeness,


$(\bar{a} \vee \bar{f} \vee \bar{h})$
$(\bar{e} \vee \bar{g} \vee h)$
$(\bar{c} \vee \bar{r} \vee f)$
$(\bar{c} \vee \bar{d} \vee e)$
$(\bar{a} \vee \bar{g} \vee \bar{e} \vee \bar{f})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{g} \vee \bar{e})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{d} \vee \bar{g})$

(a) Default clause learning


$(\bar{a} \vee \bar{f} \vee \bar{h})$
$(\bar{e} \vee \bar{g} \vee h)$
$(\bar{c} \vee \bar{r} \vee f)$
$(\bar{c} \vee \bar{d} \vee e)$
$(\bar{r} \vee d)$
$(\bar{a} \vee \bar{g} \vee \bar{e} \vee \bar{f})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{g} \vee \bar{e})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{d} \vee \bar{g})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{g})$

(b) Local minimization, with self-subsuming resolution


$(\bar{a} \vee \bar{f} \vee \bar{h})$
$(\bar{e} \vee \bar{g} \vee h)$
$(\bar{c} \vee \bar{r} \vee f)$
$(\bar{c} \vee \bar{d} \vee e)$
$(\bar{s} \vee g)$
$(\bar{d} \vee s)$
$(\bar{r} \vee d)$
$(\bar{a} \vee \bar{g} \vee \bar{e} \vee \bar{f})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{g} \vee \bar{e})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{d} \vee \bar{g})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{d} \vee \bar{s})$
$(\bar{a} \vee \bar{c} \vee \bar{r} \vee \bar{d})$
$(\bar{a} \vee \bar{c} \vee \bar{r})$

(c) Recursive minimization

Figure 4.8. Learned clause minimization for Example 4.3.5

an increase in cutoff can be used.

The idea of search restarts was shown to work effectively with clause learning [BMS00, LBMS01], and eventually became an integral part of modern CDCL SAT solvers [MMZ$^+$01, ES03, Bie08b]. More importantly, and from a theoretical perspective, it has been shown that clause learning and search restarts correspond to a proof system as powerful as general resolution, and stronger than the original

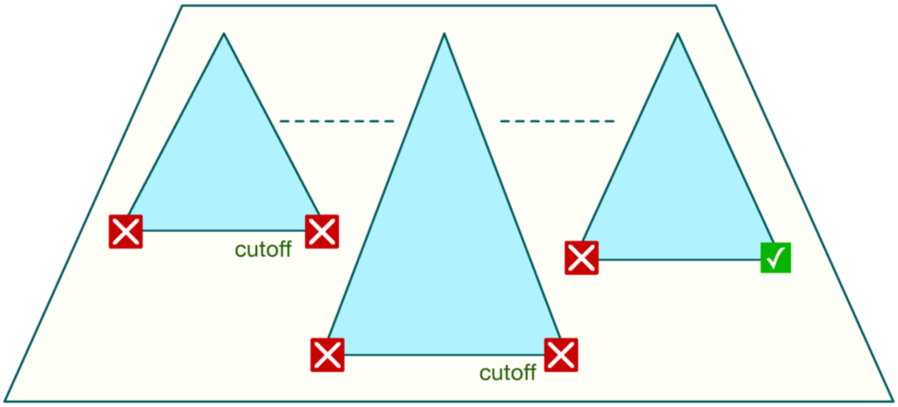


Figure 4.9. Operation of search restarts

DPLL proof system [PD09, PD11]. From a more practical perspective, branching randomization is still available in most modern CDCL SAT solvers, but mostly not used.

4.3.1.2.1. Selecting a Restart Sequence. In recent years, a number of authors proposed and evaluated different restart sequences [Hua07, Bie08b, Bie08a, SI09, BF15a]. One example are the so-called *Luby restarts* due to Luby, Sinclair and Zuckerman [LSZ93], and proposed in the CDCL SAT solver TiniSAT [Hua07]. In more recent work, different restart strategies are mixed, attempting to exploit different solver behavior between satisfiable and unsatisfiable instances [Oh15].

4.3.1.2.2. Ensuring Search Completeness. A possible limitation of search restarts is solver completeness. For example, always restarting after a fixed number of conflicts will yield an incomplete algorithm, which may be unable to solve some instances. This limitation was acknowledged in earlier work [GSK98, BMS00], and different solutions proposed. A simple solution is to increase the number of conflicts in between restarts [GSK98, BMS00]. Another solution is to ensure that enough learned clauses are kept in between restarts [LBMS01].

4.3.1.2.3. Phase Saving. A more recent technique is to keep the branching phase after the search is restarted [PD07]. Most modern CDCL SAT solvers implement the phase saving technique [Bie08a, AS09].

4.3.1.3. Deletion of Learned Clauses

From the outset [MS95, MSS96c], clause learning was proposed under the assumption that not all learned clauses would be kept, and that some of these clauses would eventually be deleted. With the improvements in performance from the early 00s [MMZ$^+$01], policies for deciding which clauses to delete became paramount [GN02]. Clauses are deleted on a regular basis after some number of conflicts. The state of the art is to use the Literal Block Distance (LBD) clause quality metric to decide which clauses to delete [AS09]. Given a learned clause, its literals are partitioned into decision levels. The number of decision levels is the LBD metric. Clauses with higher LBD metric are less likely to be relevant for future conflicts, and so can be considered for deletion. Moreover, the LBD metric has also been applied for deciding when to minimize clauses (since clauses with small LBD metric are more likely not to be deleted) [Bie13], and to decide the restart schedule [AS12, BF15a].

4.3.1.4. Conflict-Driven Branching

For a satisfiable instance, perfect branching would enable finding a satisfying assignment without conflicts, e.g. it would suffice to pick the variables and assignments from an existing model (if it were somehow known). Unfortunately, many instances are unsatisfiable and, even for the satisfiable instances, it seems fairly unlikely that a solver would be able to find the perfect way to branch.

Similarly to clause learning, which is only applied in the presence of conflicts, modern branching mechanisms aim to promote conflicts. The best known branching strategy is VSIDS (Variable State Independent Decaying Sum) [MMZ+01]. Over the years variants have been proposed [GN02], but all similar in spirit to VSIDS (and this is one critical contribution to the success of CDCL SAT solving). (See [BF15b] for a fairly recent account of branching heuristics.) The original VSIDS branching strategy can be summarized as follows:

1. A counter is associated with each variable.
2. After a clause is learned, the counters of the variables used for deriving the clause are incremented.
3. Branching consists of picking the unassigned variable with the highest counter, with ties broken randomly.
4. Counters are divided by a constant on a regular basis.

4.3.1.5. Lazy Data Structures

One additional ingredient for modern CDCL SAT solvers are lazy data structures. Modern CDCL SAT solvers can learn many clauses with a large number of literals. For these clauses, the classical data structure, associating a reference from each variable to each clause containing a literal in that variable, was effectively too slow to manage [MSS96c, MSS99]. An elegant solution is to consider lazy data structures, that do not maintain so many references [ZS00]. The most significant of these solutions is the watched literals data structure [MMZ+01].

Example 4.3.7 (Watched Literals)**.** The example in Figure 4.10 will be used as a running example to describe the operation of watched literals. The clause $\mathfrak{c}_t$ contains eight literals, named $\mathrm{A}, \mathrm{B}, \ldots, \mathrm{H}$, such that literal C is assigned value 0 at decision level 0, literal H is assigned value 0 at decision level 1, and literals B and D are assigned value 0 at decision level 2. (In general, shaded boxes denote assigned 0-valued literals.) After unit propagation at decision level 2, this clause $\mathfrak{c}_t$ is unresolved with four literals still unassigned, concretely A, D, F and G.

The watched literals data structure is organized as follows [MMZ+01]:

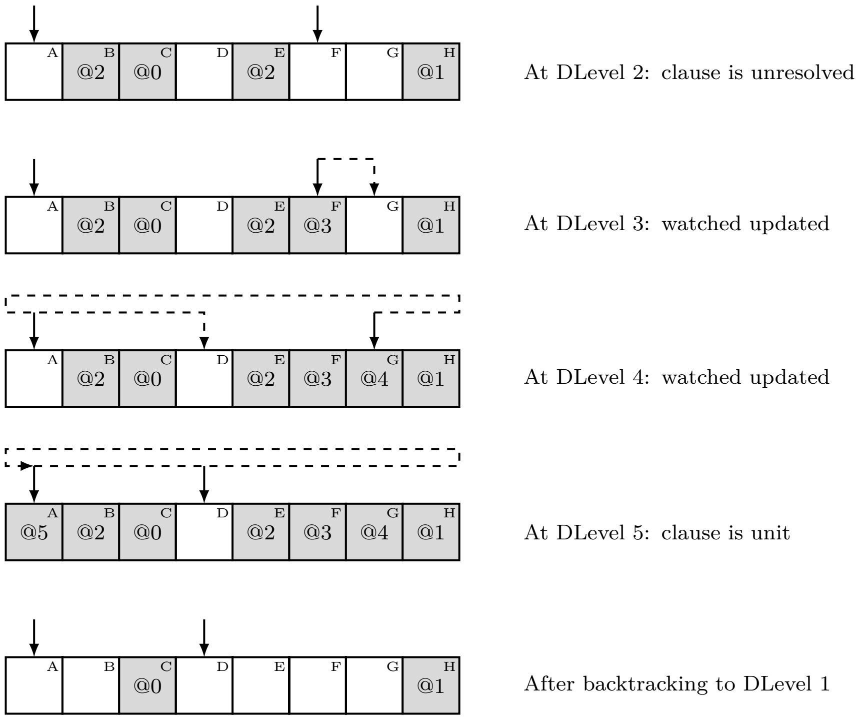

A B @2 C @0 D E @2 F G H @1
At DLevel 2: clause is unresolved
A B @2 C @0 D E @2 F @3 G H @1
At DLevel 3: watched updated
A B @2 C @0 D E @2 F @3 G @4 H @1
At DLevel 4: watched updated
A @5 B @2 C @0 D E @2 F @3 G @4 H @1
At DLevel 5: clause is unit
A B C @0 D E F G H @1
After backtracking to DLevel 1

Figure 4.10. Watched literals

1. For each clause, *exactly* two unassigned literals are watched, using two *watch references*. For the running example, at decision level 2, the watch references point to the literals A and F.
2. The clause's state is only updated when one of the watched literals is assigned. If the watched literal is satisfied, nothing more is done with respect to this clause. If the literal is falsified, then the clause is traversed in search for one non-false literal that is not already being watched. For the running example, at decision level 3, since the literal in F is assigned, then the unassigned literal in G becomes watched. At decision level 4, it is necessary to traverse most of the clause's literals to conclude that the next literal to watch is D. Finally, at decision level 5, the literal in A becomes assigned. This results in all of the clause's literals to be traversed in search for another non-watched unassigned literal. In this case, there is none, and so all literals must be traversed, and the clause is declared unit.
3. When backtracking, the watched literals information is *not* updated. The ability to leave watch references unchanged when backtracking is arguably one of the most significant features of the watched literal data structure.

When a clause becomes satisfied the simplest solution is to leave the watch references untouched. It is possible to conceive mechanisms for *parking* the watch references, such that these become unparked when the clause is no longer satisfied. Moreover, as Example 4.3.7 hints at, at each decision level, the worst-case amortized run time of updating the watched literals in a clause is linear in the clause's size. This is true if the watch references operate in circular mode (as shown in Example 4.3.7) and it is optimal [Gen13]. Some other ways of implementing watched literals (e.g. in MiniSat [ES03]) are known not to be optimal.

Some solvers implement dedicated data structures for specific types of clauses. For example, binary clauses can be represented as direct implications, which reduces not only the memory footprint, but also the overhead involved. Moreover, in many practical settings, binary clauses account for a significant percentage of the total number of clauses.

4.3.2. Additional Techniques

A number of promising techniques have been under active development in recent years, but do not yet find widespread use. An ongoing line of work, of key importance in some settings is preprocessing of CNF formulas and, more recently inprocessing [JHB12], which includes among others the blocked clause elimination technique [JBH10]. Additional recent efforts include exploiting machine learning techniques to optimize the branching heuristic [LGPC16a, LGPC16b] and more recently the restart strategy [LOM$^+$18], an improved learned clause minimization procedure [LLX$^+$17], and the use in some settings of GRASP's non-chronological backtracking strategy [NR18, MB19]. Finally, some solvers [SNC09, MHB12, Bie13] implement gaussian elimination [WvM98], with the purpose of more efficient handling of XOR constraints.

Additional low-level optimizations are often introduced in CDCL SAT solvers. The interested reader should read the available solver descriptions, e.g. from the SAT competitions [BBJS15, HS15, BHJ16, BHJ17a, BHJ17b, HJS18].

Given that problem representations are most often non-clausal [Stu13], one might expect that non-clausal CDCL SAT solvers would represent a promising direction for improving practical SAT solving. A number of efforts have been reported over the years [GeSSMS99, KGP01, MSGeS03, TBW04, DJN09]. However, the performance of non-clausal solvers has not kept up with that of clausal SAT solvers. Thus, the most commonly used solution is simply to clausify the problem representation. A similar situation was observed for other dedicated solvers operating over propositional domains [RM09].

A number of additional concepts have been the subject of a significant body of work, but such efforts have not materialized in their application in CDCL SAT solvers. Concrete examples include symmetry breaking [CGLR96, Sak09], backdoors and backbones [WGS03, KSTW05], but also phase transitions [MZK$^+$99]. Backbones denote assignments that are fixed in any model of satisfiable formulas, and find a wide range of applications. However, in practice backbones are computed with CDCL SAT solvers, and have not served to improve the performance of CDCL SAT solvers. Backdoors represent sets of variables which, for some assignement, enable a polynomial solver to satisfy the formula in polynomial time. For unsatisfiable formulas, strong backdoors are sets of variables such that, for any assignment, unsatisfiability is decided in polynomial time. Although backdoors caused significant interest, in practice backdoors can be hard to compute.

Moreover, recent results [JJ09] cast doubts in the practical effectiveness of exploiting backdoors. To our best knowledge, CDCL SAT solvers are yet to exploit backdoors. Nevertheless, recent work exploits SAT backdoors for devising cryptographic attacks [SZO$^+$18]. Finally, symmetries in SAT have been the subject of a comprehensive body of research for the last two decades. Important progress has been registered, but the performance gains of breaking symmetries can vary wildly. As a result, modern CDCL SAT solvers do not readily integrate identification of symmetry breaking predicates. Finally, phase transitions are a well-known phenomenon of combinatorial problems, denoting a transition from easy to hard problems on random instances and given some problem measure [MZK$^+$99]. Insights from phase transitions are yet to be exploited by CDCL SAT solvers for improving run times of practically-oriented formulas.

4.4. Using CDCL SAT Solvers

The practical success of CDCL SAT solvers builds upon not only a highly efficient reasoning engine, but also upon a number of supporting technologies which have witnessed very significant progress, to a large extent motivated by the success of CDCL SAT solvers. These technologies include the many ways SAT solvers have been embedded in applications, but also in a wide range of reasoners, the many ways SAT solvers are used as oracles for the class NP of decision problems, and also, the many ways in which problems can be encoded into propositional domains. This section provides a glimpse of the approaches that have been devised for using SAT solvers in practical settings. The section does not aim to be exhaustive, but instead to provide a comprehensive list of pointers from which a better understanding of the range of uses of CDCL SAT solvers can be attained.

4.4.1. Practical Use of SAT Solvers

A number of practical successes of SAT solvers depend not only on the raw performance of the solvers, but also on a number of additional features that are available in most modern implementations of CDCL SAT solvers. In many settings, SAT solvers are iteratively used as (extended) oracles for the class NP of decision problems.

4.4.1.1. CDCL SAT vs. NP Oracles

The ideal model of an NP oracle provides an answer of Yes (or No) depending on whether some decision problem has (or does not have) a solution. When using SAT solvers as oracles for the class NP, there are important differences. First, SAT solvers compute a solution (i.e. the satisfying assignment) when the answer is positive. Abstracting the actual run time, (CDCL) SAT oracles can be modeled as witness-producing oracles for NP [JaMS16]. Moreover, for negative answers, SAT solvers are often capable of providing an *unsatisfiable core*, i.e. a subset of the original set of clauses, which is by itself unsatisfiable. SAT solvers offer no guarantee in terms of the quality of the computed unsatisfiable core, e.g. one possible unsatisfiable core is the formula itself. Moreover, SAT solvers provide no guarantee in terms of the quality of the resolution proof associated with the computed unsatisfiable core. Despite this lack of guarantees in terms of the information that can be obtained given a negative outcome, and as shown in Section 4.4.3, CDCL SAT solvers find an ever increasing range of applications where both positive and negative outcomes are to be expected.

4.4.1.2. Incremental SAT with Assumptions

Most CDCL SAT solvers can be invoked given a number of assumed values on a set of variables; these are the so-called *assumptions*. In turn, this allows for incremental iterative use. Consider a clause $\mathfrak{c}_i$. If one needs to add or remove $\mathfrak{c}_i$ in between calls to a SAT solver, then the clause that is actually added to the SAT solver data structures is: $(\mathfrak{c}_i \vee \overline{s_i})$, where s_i denotes a *fresh variable* (referred to as *activation*, *selection* or *indicator* variable). As a result, a user can conditionally *program* the activation of clause $\mathfrak{c}_i$ by setting an assumption $s_i = 1$ if the clause is to be considered, or $s_i = 0$ if the clause is not to be considered. Assumptions are passed to the SAT solver, assigned and propagated before the actual search starts. CDCL SAT solvers usually assign each one of these assumption variables per decision level, and put the reference decision level immediately after all assumptions have been handled. If the SAT solver backtracks to one of these decision levels and a conflict is identified, then the SAT solver returns unsatisfiable. A simple solution to delete clauses that can be incrementally activated/deactivated is to add a unit clause $(\overline{s_i})$. Similarly, if the clause is to be declared final, meaning that it is to be used in all future calls to the SAT solver, but needs no longer to be activated/deactivated, then the solution is to add the unit clause (s_i).

Example 4.4.1. Consider a SAT solver working with clauses taken from two sets, $\mathcal{B}$ and $\mathcal{S}$:

$$\begin{aligned} \mathcal{B} &= \{(\bar{a} \vee b), (\bar{a} \vee c)\} \\ \mathcal{S} &= \{(a \vee \overline{s_1}), (\bar{b} \vee \bar{c} \vee \overline{s_2}), (a \vee \bar{c} \vee \overline{s_3}), (a \vee \bar{b} \vee \overline{s_4})\} \end{aligned}$$

$\mathcal{B}$ denotes background knowledge. These clauses are *final* and their activation *cannot* be controlled with indicator variables. In contrast, $\mathcal{S}$ denotes clauses whose activation can be conditionally selected with the indicator variables $\{s_1, s_2, s_3, s_4\}$. It can also be observed that if all clauses in $\mathcal{S}$ are activated, then the resulting formula is unsatisfiable. Nevertheless, given the set of assumptions $\{s_1 = 1, s_2 = 0, s_3 = 0, s_4 = 1\}$, the SAT solver effectively needs to decide the satisfiability of the following set of clauses:

$$\mathcal{F} = \{(\bar{a} \vee b), (\bar{a} \vee c), (a), (a \vee \bar{b})\}$$

It is immediate to conclude that $\mathcal{F}$ is satisfiable with the satisfying assignment $\{a = 1, b = 1, c = 1\}$.

Incremental SAT solving using assumptions was first proposed in the MiniSat SAT solver [ES03], and since the mid 00s it has been applied in most applications where CDCL SAT solvers are used as (witness-producing) oracles. Incremental SAT solving with assumptions has enabled solving a growing number of problems, for which the SAT solver is used as an oracle. Section 4.4.3 provides

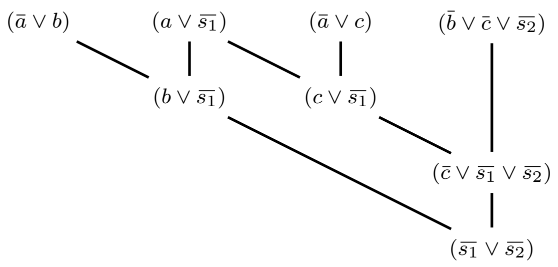

$(\bar{a} \vee b)$
$(a \vee \overline{s_1})$
$(\bar{a} \vee c)$
$(\bar{b} \vee \bar{c} \vee \overline{s_2})$
$(b \vee \overline{s_1})$
$(c \vee \overline{s_1})$
$(\bar{c} \vee \overline{s_1} \vee \overline{s_2})$
$(\overline{s_1} \vee \overline{s_2})$

Figure 4.11. Finding unsatisfiable cores

a glimpse of the many applications where incremental SAT with assumption has been applied over the last decade. Moreover, a fairly recent topic of research is on improving the use of assumptions in such settings [LB13, ALS13].

Although the incremental SAT approach outlined in this section is standard, there have been recent alternative approaches to implement incremental SAT [NR12, NRS14, FBS19].

4.4.1.3. Unsatisfiable Core Extraction

In a number of practical settings one is often interested in understanding the causes of unsatisfiability. An often used representation of the causes of unsatisfiability are *unsatisfiable cores* (or unsatisfiable subsets), which denote subsets of a CNF formula which are by themselves unsatisfiable. For CDCL SAT solvers with an incremental interface, unsatisfiable cores can be computed by checking the assumptions that remain in the final learned clause.

For unsatisfiable formulas, assuming no indicator variables are in use, a SAT solver must eventually learn the empty clause, as the final confirmation that the formula is unsatisfiable. If indicator variables are in use there are a few important differences to note. Concretely, any indicator variable is necessarily transferred to a learned clause. As a result, for unsatisfiable formulas, instead of learning the empty clause, the SAT solver terminates by learning a clause that contains *all* the indicator variables of clauses used for proving (by resolution) the unsatisfiability of the formula. Clearly, it is easy to argue that the indicator variables in the final learned clause *must* be a superset of the clauses necessary to prove unsatisfiability by resolution.

Example 4.4.2. Consider again the sets of clauses $\mathcal{B}$ and $\mathcal{S}$ from Example 4.4.1. Moreover, given the assumptions $\{s_1 = 1, s_2 = 1, s_3 = 1, s_4 = 1\}$, an unsatisfiable core obtained by a CDCL SAT solver is represented by the indicator variables $\{s_1, s_2\}$. This signifies that the clauses associated with this set of indicator variables are inconsistent with the background knowledge. An explanation for the operation of the CDCL SAT solver is shown in Figure 4.11.

4.4.1.4. Proof Tracing

Applications of SAT solvers in safety-critical applications, or in computer-assisted proofs, motivate the need for SAT solvers to have their results validated. The validation of positive (i.e. satisfiable) outcomes is straightforward. (This is a natural consequence of SAT being in the class NP, and so having succinct certificates.) Indeed, for a positive outcome, it suffices to validate the satisfying assignment against the CNF formula, which can be done in linear time on the size of the formula. In contrast, deciding unsatisfiability is in coNP, and so the existence of succinct certificates is unlikely. Interest in validating unsatisfiable outcomes can be traced at least to the early 00s [VG02], with the approaches now in common use first outlined in 2003 [ZM03, GN03]. In these approaches, CDCL SAT solvers are instrumented to output a trace of their execution, which can then be checked by validating the associated resolution proof. Existing approaches for checking proofs of unsatisfiability can rely on ad-hoc implementations (referred to as *proof checking*) or on implementations extracted from formally verified checkers (referred to as *proof certification*). A wealth of proof tracing formats have been proposed over the years for checking unsatisfiability proofs [Bie08b, Van08, Van12, HHW13, HHJW13, WHHJ14, HHJW14, CFMSSK17, CFHHJ$^+$17], which evolved to enabling proof systems stronger than resolution, but also with recent formats aiming at efficient proof certification. Work on devising proof certification approaches has also witnessed remarkable progress over the last decade [Web06, WA09, DFMS10, WHHJ13, CFMSSK17, CFHHJ$^+$17, HHJKW17].

The importance of checking (and of certifying) proofs of unsatisfiability cannot be overstated. For example, in recent years, solvers participating in the SAT competition are required to output traces in the case of unsatisfiable outcomes, to enable proof checking (and certification) [BHJ17a]. Furthermore, proof checking and certification has been at the core of computer-assisted mathematical proofs [KL14, KL15, HKM16, HK17, Heu18].

4.4.1.5. Warm Restarts

For iterative uses of SAT solvers, when the run time of each call to a SAT solver can be expected to be negligible, and in settings where the search tree has many levels, one technique to reduce the run time is to consider *warm restarts*[8]. The idea is to bootstrap the SAT solver with the final set of decisions from the last (positive) call to the SAT solver [JLSS14, PIMMS15, Pre15].

4.4.2. Modeling with Propositional Logic

The success of CDCL SAT solvers also motivated the quick growth of problem solving by encoding problems into propositional logic. Over the last two decades, a large number of novel propositional encodings have been proposed, which are used in an ever increasing range of applications. This section briefly overviews recent work on modeling with propositional logic.

Propositional logic is a fairly low level knowledge representation formalism. The following sections provide brief accounts of the approaches used to represent complex constraints with propositional logic. Well-known examples include cardinality constraints, pseudo-Boolean constraints, but also solutions for devising propositional encodings for more expressive domains.

[8]To our best knowledge, the term was coined by A. Biere around 2012.

For constraints defined on propositional variables, the most often used measure of quality of propositional encodings is whether arc-consistency in the original constraint can be achieved by unit propagation [Gen02, ES06, RM09]. When this is the case, the propositional encoding is said to *ensure* arc-consistency. The arc-consistency measure of quality is tightly related with the ability of unit propagation to reveal inconsistent assignment as early as possible. For constraints defined on finite domains a often used measure of quality is propagation completeness [BKN+09, BMS12].

4.4.2.1. Linear Inequalities

Linear inequalities over propositional variables are of the form:

$$\sum_{i=1}^{n} a_i x_i \bowtie b \tag{4.7}$$

where $\bowtie \in \{<, \leq, =, \geq, >\}$, and where in the general case $a_i, b \in \mathbb{R}$. A well-known encoding of linear inequalities into propositional logic was proposed by Warners [War98]. This encoding is linear on the size of the input representation (which can be exponential on the number of variables), but does not ensure arc-consistency. In SAT solving, linear inequalities are often organized into cardinality constraints and pseudo-Boolean constraints. For each, there are encodings which ensure arc-consistency, with worst-case polynomial representations on the size of the input representation. There are a few additional kinds of linear inequalities studied in recent years. The next sections briefly overview the different kinds of linear inequalities and their encodings. More detailed accounts can be found in standard references [Pre09, RM09].

4.4.2.1.1. Encoding Cardinality Constraints. One of the most often used types of constraints are cardinality constraints of the form:

$$\sum_{i=1}^{n} x_i \bowtie k \tag{4.8}$$

where $\bowtie \in \{<, \leq, =, \geq, >\}$. A well-known special case is $k = 1$, with specific often used constraints being referred to as AtMost1, AtLeast1 and Equals1. Clearly, the case AtLeast1 can be encoded with a single clause, and so the interesting case is the encoding of AtMost1 constraints.

For the general case of cardinality constraints, some of the best known encodings include:

1. Sorting networks [Bat68, ES06].
2. Totalizers [BB03].
3. Cardinality networks [ANORC09, ANORC11b].
4. Pairwise cardinality networks [CZI10].
5. Modulo totalizer [OLH+13].

One can also consider a simple generalization of the pairwise encoding for AtMost1 constraints, but the encoding size is exponential in n and k. There is also recent work on encodings that ensure arc-consistency [Kar18].

In addition to the encodings above, well-known encodings for the AtMost1 constraint include:

1. Ladder/regular [GN04, AM04].
2. Bitwise [FP01, Pre07].
3. And the straightforward pairwise encoding [Pre09].

All of the above encodings ensure arc-consistency.

4.4.2.1.2. Conditional & Soft Cardinality Constraints. In some settings cardinality constraints may be conditional, i.e. of the form:

$$y_j \to \left(\sum_i x_i \bowtie k \right) \tag{4.9}$$

depending on the value of y_j, the cardinality must (or needs not) hold. Conditional cardinality have been proposed and used in different settings, including to represent soft cardinality constraints [MDMS14] in MaxSAT, in bounded path finding [EN15], and for computing the max clique in undirected graphs [IMMS17a]. Recent work investigates mechanisms to ensure arc-consistency of conditional cardinality constraints [BJRS18].

4.4.2.1.3. Encoding Pseudo-Boolean Constraints. The most general form of linear inequalities, i.e. the so-called Pseudo-Boolean (PB) constraints can be framed as follows [RM09]:

$$\sum_{i=1}^{n} a_i l_i \geq b \tag{4.10}$$

where $l_i \in \{x_i, \overline{x_i}\}$, $x_i \in \{0, 1\}$, $a_i, b \in \mathbb{N}_0$.

Motivated by the applications of CDCL SAT solvers, and problem encodings into SAT, there has been recent work on finding encodings for PB constraints which ensure arc-consistency. Concrete examples include:

1. Binary decision diagrams (BDDs) [ES06], with the caveat of requiring a worst-case exponential representation on the size of the PB constraint representation.
2. Polynomial watchdog encoding [BBR09] which, to our best knowledge, was the first polynomial encoding of PB constraints that ensures arc-consistency.
3. Recent improvement to the polynomial watchdog encoding [ANORC11a, ANO$^+$12, MPS14].

Besides the operational encoding of Warners [War98], other encodings have been proposed which do not ensure arc-consistency [ES06, MDMS14]. Similar to soft cardinality constraints, weighted boolean optimization [MMSP09] proposed the use of soft PB constraints.

4.4.2.2. Encoding More Expressive Domains

There has been extensive work on translating more expressive domains into SAT, to be solved with CDCL SAT solvers. Two well-known examples are planning and (hardware) model checking [KS99, BCCZ99, BCC$^+$99, Rin09]. Another well-known example is CSP [Wal00, Gen02], with a number of tools implementing translations of CSP into SAT. There have been efforts to encode problems in NP to SAT, e.g. NP-Spec [CS05]. Other examples include the family of relational model finders [Jac00, TJ07], but also encodings of programming languages into SAT [CKL04, Kro09].

4.4.3. SAT Solvers as Witness-Producing Oracles

The last decade has witness the quick growth of problem solving where oracles for NP are replaced with CDCL SAT oracles. As mentioned earlier, CDCL SAT solvers extend the notion of NP oracle by producing witnesses in the case of positive outcomes, and (compact) explanations (i.e. unsatisfiable cores) in the case of negative outcomes [JaMS16]. This section overviews this ongoing work. The section aims at illustrating the range of ideas being developed, but not at being exhaustive, both in the topics and in the references. The interested reader can find additional references in the references cited.

4.4.3.1. Finding Cardinality-Minimal Subsets

The identification of cardinality-minimal subsets finds many practical applications. One well-known example is Maximum Satisfiability (MaxSAT), where the goal is to compute a cardinality maximal set of constraints that is consistent. The problem is of interest when one starts from an inconsistent (i.e. unsatisfiable, often referred to as overconstrained) formula. The practical importance of MaxSAT is illustrated by recent surveys [MHL$^+$13, SZGN17], covering a wide range of applications. Initial practical algorithms for MaxSAT focused on branch-and-bound search [LM09]. This state of affairs changed in 2006, with the FM core-guided algorithm for MaxSAT [FM06]. Core-guided MaxSAT algorithms are based on iteratively calling a CDCL SAT solver, finding unsatisfiable cores, and using relaxation variables and cardinality constraints for finding the least cost way of relaxing the CNF formula so as to recover consistency. Since the FM algorithm, a wealth of core-guided MaxSAT algorithms have been devised [MSP07, MMSP09, ABL09, ABL13, IMM$^+$14, MJML14, MDMS14]. These include the now widely used MSU3 algorithm [MSP07] and variants [ABL13, MJML14], improvements to the based FM algorithms [MMSP09, ABL09], and the use of soft-cardinality constraints [MDMS14]. A more detailed account of earlier core-guided MaxSAT algorithms is included in a recent survey [MHL$^+$13]. A different MaxSAT approach that exploits CDCL SAT solvers consists in the iterative computation of minimum hitting sets [DB11, SBJ16]. The computed minimum hitting sets identify a set of clauses to exclude. The rest is checked for satisfiability with a (CDCL) SAT solver. If the formula is satisfiable, then the result is the MaxSAT solution, otherwise a computed unsatisfiable core is added to the set of sets for which a minimum hitting set is to be computed, and the process is repeated.

Another example of computing cardinality-minimal subsets is minimal satisfiability (MinSAT) [KKM94]. As with MaxSAT, initial algorithms focused on branch-and-bound search [LMQZ10]. The use of CDCL SAT solvers, as oracles for

NP, for this problem was proposed by different researchers [ALMZ12, ZLMA12, HMPMS12, IMPMS16].

4.4.3.2. Finding Subset-Minimal Subsets

In some settings one is interested in computing subset-minimal subsets. A well-known example is the extraction and enumeration of minimal unsatisfiable subsets (MUSes)[GMP08, MS10, Nad10, MSL11, BLMS12, NRS13, MSJB13, BK15, NBMS18], or in more general terms, to find minimal explanations for over-constrained systems of constraints. Another example is the extraction and enumeration of minimal correction subsets (MCSes) and their complement: maximal satisfiable subsets (MSSes) [MSHJ$^+$13, GLM14a, BDTK14, MPMS15, MIPMS16]. For both MUSes and MCSes, the state of the art algorithms are based on iterative calls to a SAT oracle, with a number of key practical optimizations developed in recent years.

There are many other examples of computing subset-minimal subsets, that exploit CDCL SAT solvers as oracles. Most of these can be cast as instantiations of the Minimal Set over Monotone Predicate (MSMP) problem [MSJB13, MSJM17]. Additional well-known examples include finding minimal models [SI10, SBTLB17], computing the autarky of an unsatisfiable CNF formula [MSIM$^+$14, KMS15] and identifying the backbone (i.e. the fixed assignments) of a satisfiable formula [MSJL10, ZWM11, ZWSM11, JLMS15]. Recent examples include [GLM14b, GL15, BGL15, IMMV16], among many others. MSMP has also been exploited in settings where monotonicity does not hold [BIV18].

4.4.3.3. Solving Decision Problems Beyond NP

SAT oracles have achieved significant success in solving decision problems beyond NP. Arguably the most visible have been the stream of improvements made to QBF solvers, by exploiting different forms of abstraction refinement, and also of SAT solvers as oracles [JMS11, JKMSC12, JMS15, RT15, JKMSC16, RS16]. Moreover, the development of an exact pure SAT-based implementation of two-level logic minimization [IPMS15] resulted in an effective alternative to the Quine-McCluskey algorithm [Qui52, Qui55, McC56], the reference two-logic level minimization algorithm for more than five decades.

Abstraction refinement and SAT oracles have been applied to modal logic reasoners [CLLB$^+$17, LLBdLM18], but also to propositional abduction [SWJ16, IMMS16] and quantified QBF with soft clauses (QMaxSAT) [IJMS16], among others.

4.4.3.4. Enumerating Sets & Solutions

SAT oracles have enabled observable success in the enumeration of explanations and relaxations of overconstrained systems of constraints. This is the case with the enumeration of minimal unsatisfiable subsets [LPMMS16], the enumeration of minimal correction subsets [PMJaMS18, GIL18], but also the enumeration of prime implicants and implicates [JMSSS14, PIMMS15]. A related problem of practical interest is to find the union of all sets, e.g. MUSes or MCSes. One concrete approach is HuMUS [NBE12].

4.4.3.5. Approximate & Exact Solution Counting

Model counting finds a number of important applications. Early work focused on adding dedicated techniques to CDCL SAT solvers to improve exact counting. Concrete examples include connected components and component caching [BJP00, SBK05, GSS09].

More recent work focused on approximate counting, using off-the-shelf CDCL SAT solvers as oracles [EGS12, CMV13a, CMV13b, CMV16, XLE$^+$16].

4.4.3.6. Portfolios

The development of portfolios of reasoners [XHHLB08, LBHHX14] has been an active area of research, finding applications for different computational problems. In the case of portfolios for SAT, CDCL SAT solvers represent an integral part of the panoply of solvers that can be used.

4.5. Impact of CDCL SAT Solvers

Over the last two decades, CDCL SAT solving technology has been truly disruptive. This section summarizes the impact of CDCL SAT solvers in terms of well-known practical applications, but also from a theoretical perspective.

4.5.1. Practical Successes

CDCL SAT solvers have been a disruptive technology since their inception, having replaced, or at least complemented, well-established alternative technologies. A number of well-known applications exist but, more importantly, CDCL SAT solvers are in a growing number of cases the engines' engine, i.e. for a number of reasoners in different settings, the most promising approach uses a CDCL SAT solver or mimics the organization of CDCL SAT solvers. This section provides a brief glimpse at the growing range of applications of CDCL SAT solvers.

A few of the best-known hallmark applications of CDCL SAT solvers include:

1. Bounded model checking [BCCZ99, BCC$^+$99, Bie09].
2. Unbounded model checking [McM03, Bra11, McM18].
3. Software model checking [CKL04, Kro09].
4. Automated planning [KS99, Rin09].
5. Computer-assisted mathematical proofs [KL14, KL15, HKM16, HK17].

Recent strategic applications of CDCL SAT solvers involve applications in explainable machine learning [IPNaMS18, NIPMS18].

CDCL SAT solvers are also the main engine for a number of reasoners, or have strongly influenced their design. Concrete examples include:

1. MaxSAT (see Section 4.4.3.1)
2. QBF (see Section 4.4.3.3).
3. Modal logics(see Section 4.4.3.3).
4. Satisfiability Modulo Theories (SMT) [BSST09].
5. Answer Set Programming (ASP) [GKKS12].

6. Lazy Clause Generation (LCG) in constraint programming [OSC09].
7. Theorem proving, both with instantiation-based methods [GK03, Kor08], but more recently with superposition theorem provers [KV13, Vor14].

The general setting of model based diagnosis [Rei87], which can be viewed as the problem of explaining overconstrained systems of constraints, finds a wealth of key practical applications [Jun04, BS05, LS08, MSKC14], for which CDCL SAT solving is becoming a key instrument [MSKC14, MSJIM15, AMMS15, LPMMS16].

4.5.2. Theoretical Significance

Recent years have witnessed significant progress in understanding CDCL SAT solvers from a propositional proof complexity perspective. Among the ideas that compose CDCL SAT solvers, from a proof complexity perspective, the key aspects of CDCL SAT solvers are clause learning and search restarts. Whereas the basic backtrack search SAT algorithm (DPLL) is associated with tree-like resolution [BKS04], a proof system weaker than resolution, CDCL SAT solving corresponds instead to a proof system as strong as general resolution [PD09, PD11, AFT09, AFT11]. These recent results build on the clause learning procedure used by CDCL SAT solvers [MS95, MSS96c], namely on the fact that learned clauses are *1-empowering*, and on the use of search restarts [GSC97, GSK98]. A more recent result shows that with a suitable preprocessing technique, restarts can be simulated on a clause learning SAT solver [BS14].

Motivated by the remarkable performance achieved by CDCL SAT solvers, researchers have investigated whether efficient algorithms for stronger proof systems could be devised. This has been the case of extended resolution [Hua10, AKS10], cutting planes [LBP10, EN18, EGCNV18], a recently proposed (dual-rail) MaxSAT-based proof system [IMMS17b, BBI^{+}18], but also the recently proposed proof systems related with proof tracing [HKSB17, KRPH18].

Although improvements to CDCL SAT solvers continue to be reported, one would expect far more significant impact if practically effective algorithms could be devised for proof systems stronger than general resolution. Although new insights have been reported, it is unclear the success of SAT can be replicated for such stronger proof systems.

4.6. Historical Perspective

The name Confict-Driven Clause Learning (CDCL) can be traced to L. Ryan's 2004 MSc thesis [Rya04] to designate SAT solvers implementing the DPLL algorithm, but extended with clause learning, search restarts, conflict-aware branching and lazy data structures.

From a theoretical perspective, the most significant contributions of CDCL SAT solvers are clause learning and search restarts.

The clause learning technique used in modern CDCL SAT solvers was proposed in the mid 90s [MS95, MSS96c], with a more extensive treatment in the late 90s [MSS99]. Later work also proposed and used related ideas [BJS97, Zha97].

Nevertheless, all modern CDCL SAT solvers implement the clause learning mechanism first detailed in [MS95, MSS96c], namely by exploiting decision levels and UIPs when learning clauses. The relationship between implication graphs, decision levels, and non-chronological backtracking, without clause learning, was actually described in earlier work [MSS93, MSS94].[9] In more general settings, learning from conflicts can be traced to the 70s [SS77]. Ideas on backtracking non-chronologically can be traced to about the same time [Gas77, Gas79]. A few additional relevant later works exist [Dec90, Gin93, Pro93]. Nevertheless, and to our best knowledge, in no other area has learning had as significant an impact as in SAT.

The idea of exploiting dominators in conflict analysis is tightly related with the concept of unique sensitization points (USPs) in ATPG [FS83, SA89, MSS94]; both concepts exploit dominators, from which additional inferences can be made. USPs are (static or dynamic) dominators of a circuit with respect to the fault location. UIPs are also dominators, but related with implication sequences. From the inception of UIPs [MS95, MSS96a, MSS96b, MSS96c, MSS99], the name aimed to acknowledge the relationship with USPs.

Search restarts were first proposed in the context of SAT in 1998 [GSC97, GSK98], with the specific purpose of improving the behavior of DPLL-like SAT solvers on satisfiable instances. The practical benefits of exploiting search restarts with clause learning were first documented in 2000 [BMS00], and are now used in any CDCL SAT solver [MMZ$^+$01, ES03, Bie08b, AS09]. More recent work offered a theoretical justification for the use of clause learning and search restarts [PD09, PD11, AFT11], showing that it corresponds to a proof system as powerful as general resolution. The concept of *empowering implicate* computed by clause learning SAT solvers [MS95, MSS96c] is crucial for these results.

From a practical perspective, the most significant contributions of CDCL SAT solvers include not only clause learning and search restarts, but also conflict-driven branching and lazy data structures. All these contributions are crucial for the performance of modern CDCL SAT solvers.

The idea of conflict-driven branching can be traced at least to the late 90s [MS99], where the branching heuristics DLIS and DLCS, by dynamically exploiting literal counts, would take into account the literals in already learned clauses. Both DLIS and DLCS were implemented in the GRASP SAT solver since the mid 90s [MS95, MSS96c]. The importance of conflict-driven branching was demonstrated in the Chaff SAT solver [MMZ$^+$01], with the VSIDS branching heuristic, which became the de facto reference branching heuristic ever since. Indeed, and even though alternatives have been proposed over the years, VSIDS-like branching is still used in most modern CDCL SAT solvers.

The origins of exploiting lazy data structures in SAT solvers can be traced to 2000 [ZS00], concretely the idea of maintaining head and tail literals in clauses. The much more efficient alternative of watched literals was proposed in the Chaff SAT solver [MMZ$^+$01]. With minor modifications, watched literals are still used in modern CDCL SAT solvers.

[9] The name *implication graph* was borrowed from [Lar89, Lar92], where it was used with a different meaning, and which is closely related to the directed graphs associated with 2CNF formulas used in the work Aspvall, Plass and Tarjan [APT79].

In recent times, CDCL is used to mean SAT solvers that implement clause learning, search restarts, conflict-driven branching and lazy data structures, but also a number of commonly used additional techniques. These include phase saving [PD07] but also literal blocks distance [AS09].

4.7. To Probe Further

The chapter provides a comprehensive list of references on the implementation but also one the practical uses of CDCL SAT solvers. Many CDCL SAT solvers have been implemented over the years. The following list summarizes those that are currently publicly available and that have impacted the organization of CDCL SAT solvers.

1. GRASP [MSS96c, MSS99][10] was the first SAT solver to implement conflict analysis as used in modern CDCL SAT solvers, including clause learning with UIP identification and non-chronological backtracking. Search restarts and its interaction with clause learning were introduced in GRASP at a later stage [BMS00, LBMS01]. GRASP has been open source since its inception.
2. Chaff [MMZ+01, ZMMM01][11] introduced efficient lazy data structures, the VSIDS branching heuristic, and also proposed stopping conflict analysis at the first UIP. Chaff is open source, although some of its ideas have been patented [MMM08][12].
3. MiniSat [ES03][13] introduced a simple architecture for the organization of CDCL SAT solvers, one that is still used by most modern CDCL SAT solvers. MiniSat also introduced learned clause minimization in a 2005 release, but this was documented only at a later stage [SB09]. MiniSat has been open source since its inception.
4. Since the mid 00s, A. Biere has developed a number of state of the art SAT solvers, including Lingeling, PicoSAT, PrecoSAT and Limmat[14]. These solvers are open source, and some are overviewed in [Bie08b, Bie13, BF15a, Bie16, Bie17].
5. Another CDCL SAT solver is Glucose [AS09][15], which introduced the LBD metric and proposed ways for using this metric in practice. Glucose builds on the MiniSat code base and it is also open source.
6. A more recent CDCL SAT solver is MapleSAT [LGPC16b, LGPC16a, LOM+18].[16] MapleSAT build on the Glucose code base, and it is open source.
7. There are SAT solvers that exploit specific kinds of reasoning. One example is CryptoMiniSat[17] [SNC09]. CryptoMiniSat is open source.

[10] GRASP is available from `https://sites.google.com/site/satgrasp/`.

[11] Chaff is available from `https://www.princeton.edu/~chaff/software.html`.

[12] There have been a number of patents related with CDCL SAT solving and applications. One additional example is related with the work on the BerkMin SAT solver [GN08].

[13] MiniSat is available from github: `https://github.com/niklasso/minisat.git`.

[14] Some of A. Biere's SAT solvers are available from: `http://fmv.jku.at/software/`.

[15] Glucose is available from: `http://www.labri.fr/perso/lsimon/glucose/`.

[16] MapleSAT is available from `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/`.

[17] Available from `https://github.com/msoos/cryptominisat`.

Furthermore, the SAT competitions[18] are an invaluable source of information about state of the art CDCL SAT solvers [BBJS15, HS15, BHJ16, BHJ17a, BHJ17b, HJS18].

A recent effort towards simplifying the implementation of SAT-based tools is the PySAT project[19] [IMMS18]. PySAT offers a Python API to the low-level operation of SAT solvers, as described in previous sections. In turn, this has enabled PySAT-based tools to achieve state of the art performance in a number of settings [IMMS18]. Besides a Python API for interfacing a number of SAT solvers, PySAT offers a variety of encodings of cardinality constraints, which can be helpful for encoding problems into SAT. Furthermore a number of well-known algorithms for solving different function problems are provided as examples of use. These include MUS and MCS extraction, but also MaxSAT solving. The PySAT project is open source, with an MIT license, and should be straightforward to install on most Linux and MacOS distributions.

4.8. Conclusions & Research Directions

This chapter provides an overview of the implementation and practical uses of CDCL SAT solvers. CDCL SAT solving represents one concrete example of remarkable success at solving NP-complete decision problems in practice, and defies the often assumed worst-case characterizations of the run-time of SAT solvers. Regarding the success of practical SAT solving, one of the most striking aspects of this success is that for many years SAT was perceived as an extremely hard problem; all this changed in the mid 90s and early 00s. Similar successes have been witnessed in other areas, including integer linear programming.

As the chapter hopes to demonstrate, there has been a steady stream of improvements to what represents a CDCL SAT solver. Progress is expected to continue. Performance improvements continue to be attained. The application of machine learning has contributed to optimizing components of CDCL SAT solvers, but has also served to develop very effective portfolios of SAT solvers. New insights at the intersection of SAT solver technology and ML are to be expected. The fast pace at which CDCL SAT oracles are being used for problem solving, in different domains, suggests that one should expect that a much wider range of successes of CDCL SAT solving will be witnessed in the near future.

Acknowledgments. Armin Biere, Alexey Ignatiev, and Antonio Morgado provided comments to earlier drafts of this document. Donald Knuth provided detailed comments to the first edition of this chapter.

References

[ABH+08] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008.

[18]Detailed information about the SAT competitions since 2002 is available from `https://www.satcompetition.org/`. This website also contains links to the SAT Races of 2006, 2008 and 2018, and the SAT Challenge of 2012.

[19]PySAT is available from `https://pysathq.github.io/`.

[ABL09] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In *SAT*, pages 427–440, 2009.

[ABL13] C. Ansótegui, M. L. Bonet, and J. Levy. SAT-based MaxSAT algorithms. *Artif. Intell.*, 196:77–105, 2013.

[AFT09] A. Atserias, J. K. Fichte, and M. Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 114–127. Springer, 2009.

[AFT11] A. Atserias, J. K. Fichte, and M. Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.*, 40:353–373, 2011.

[AKS10] G. Audemard, G. Katsirelos, and L. Simon. A restriction of extended resolution for clause learning SAT solvers. In *AAAI*. AAAI Press, 2010.

[ALMZ12] C. Ansótegui, C. M. Li, F. Manyà, and Z. Zhu. A SAT-based approach to MinSAT. In *CCIA*, volume 248 of *Frontiers in Artificial Intelligence and Applications*, pages 185–189. IOS Press, 2012.

[ALS13] G. Audemard, J.-M. Lagniez, and L. Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *SAT*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer, 2013.

[AM04] C. Ansótegui and F. Manyà. Mapping problems with finite-domain variables to problems with boolean variables. In *SAT*, pages 1–15, 2004.

[AMMS15] M. F. Arif, C. Mencía, and J. Marques-Silva. Efficient MUS enumeration of horn formulae with applications to axiom pinpointing. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 324–342. Springer, 2015.

[ANO$^+$12] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. A new look at BDDs for pseudo-boolean constraints. *J. Artif. Intell. Res.*, 45:443–480, 2012.

[ANORC09] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In *SAT*, pages 167–180, 2009.

[ANORC11a] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. BDDs for pseudo-boolean constraints - revisited. In *SAT*, pages 61–75, 2011.

[ANORC11b] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.

[AS09] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.

[AS12] G. Audemard and L. Simon. Refining restarts strategies for SAT and UNSAT. In M. Milano, editor, *CP*, pages 118–126, 2012.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[BB03] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *CP*, pages 108–122, 2003.

[BBI$^+$18] M. L. Bonet, S. Buss, A. Ignatiev, J. Marques-Silva, and A. Morgado. MaxSAT resolution with the dual rail encoding. In *AAAI*, pages 6565–6572. AAAI Press, 2018.

[BBJS15] A. Balint, A. Belov, M. Järvisalo, and C. Sinz. Overview and analysis of the SAT challenge 2012 solver competition. *Artif. Intell.*, 223:120–155, 2015.

[BBR09] O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of pseudo-boolean constraints into CNF. In *SAT*, pages 181–194, 2009.

[BCC$^+$99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC*, pages 317–320. ACM Press, 1999.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BDTK14] F. Bacchus, J. Davies, M. Tsimpoukelli, and G. Katsirelos. Relaxation search: A simple way of managing optional clauses. In *AAAI*, pages 835–841. AAAI Press, 2014.

[BF15a] A. Biere and A. Fröhlich. Evaluating CDCL restart schemes. In *Sixth Pragmatics of SAT workshop*, 2015.

[BF15b] A. Biere and A. Fröhlich. Evaluating CDCL variable scoring schemes. In *SAT*, pages 405–422, 2015.

[BGL15] P. Besnard, É. Grégoire, and J.-M. Lagniez. On computing maximal subsets of clauses that must be satisfiable with possibly mutually-contradictory assumptive contexts. In *AAAI*, pages 3710–3716. AAAI Press, 2015.

[BHJ16] T. Balyo, M. J. H. Heule, and M. Järvisalo. Proceedings of SAT competition 2016. 2016.

[BHJ17a] T. Balyo, M. J. H. Heule, and M. Järvisalo. SAT competition 2016: Recent developments. In *AAAI*, pages 5061–5063. AAAI Press, 2017.

[BHJ17b] T. Balyo, M. H. Heule, and M. Järvisalo. Proceedings of SAT competition 2017: Solver and benchmark descriptions. 2017.

[BHvMW09] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bie08a] A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008.

[Bie08b] A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[Bie09] A. Biere. Bounded model checking. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.

[Bie13] A. Biere. Lingeling, plingeling and treengeling entering the SAT competition 2013. *Proceedings of SAT competition*, 51, 2013.

[Bie16] A. Biere. Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. *Proc. of SAT Competition*, pages 44–45, 2016.

[Bie17] A. Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2017. *SAT COMPETITION 2017*, page 14, 2017.

[BIV18] R. Berryhill, A. Ivrii, and A. G. Veneris. Finding all minimal safe inductive sets. In *SAT*, pages 346–362, 2018.

[BJP00] R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *AAAI*, pages 157–162, 2000.

[BJRS18] A. Boudane, S. Jabbour, B. Raddaoui, and L. Sais. Efficient sat-based encodings of conditional cardinality constraints. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 181–195. EasyChair, 2018.

[BJS97] R. J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI*, pages 203–208, 1997.

[BK15] F. Bacchus and G. Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2015.

[BKN$^+$09] C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh. Decompositions of all different, global cardinality and related constraints. In *IJCAI*, pages 419–424, 2009.

[BKS04] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.

[BL99] H. K. Büning and T. Lettmann. *Propositional logic: deduction and algorithms*. Cambridge University Press, 1999.

[BLMS12] A. Belov, I. Lynce, and J. Marques-Silva. Towards efficient MUS extraction. *AI Commun.*, 25(2):97–116, 2012.

[BMS00] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 489–494. Springer, 2000.

[BMS12] L. Bordeaux and J. Marques-Silva. Knowledge compilation with empowerment. In *SOFSEM*, volume 7147 of *Lecture Notes in Computer Science*, pages 612–624. Springer, 2012.

[Bra11] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[BS05] J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186, 2005.

[BS14] P. Beame and A. Sabharwal. Non-restarting SAT solvers with simple preprocessing can efficiently simulate resolution. In *AAAI*, pages 2608–2615. AAAI Press, 2014.

[BSST09] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[CESS08] K. Claessen, N. Een, M. Sheeran, and N. Sorensson. Sat-solving in practice. In *WODES*, pages 61–67. IEEE, 2008.

[CFHHJ$^+$17] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt Jr., M. Kaufmann, and P. Schneider-Kamp. Efficient certified RAT verification. In *CADE*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017.

[CFMSSK17] L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp. Efficient certified resolution proof checking. In *TACAS (1)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135, 2017.

[CGLR96] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159. Morgan Kaufmann, 1996.

[CKL04] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CLLB$^+$17] T. Caridroit, J.-M. Lagniez, D. Le Berre, T. de Lima, and V. Montmirail. A SAT-based approach for solving the modal logic S5-satisfiability problem. In *AAAI*, pages 3864–3870, 2017.

[CMV13a] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable and nearly uniform generator of SAT witnesses. In *CAV*, pages 608–623, 2013.

[CMV13b] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *CP*, pages 200–216, 2013.

[CMV16] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *IJCAI*, pages 3569–3576, 2016.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[CS05] M. Cadoli and A. Schaerf. : Compiling problem specifications into SAT. *Artif. Intell.*, 162(1-2):89–120, 2005.

[CZI10] M. Codish and M. Zazon-Ivry. Pairwise cardinality networks. In *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 2010.

[DB11] J. Davies and F. Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *CP*, pages 225–239, 2011.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.

[DFMS10] A. Darbari, B. Fischer, and J. Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In *ICTAC*, volume 6255 of *Lecture Notes in Computer Science*, pages

260–274. Springer, 2010.

[DJN09] R. Drechsler, T. A. Junttila, and I. Niemelä. Non-clausal SAT and ATPG. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 655–693. IOS Press, 2009.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[EGCNV18] J. Elffers, J. Giráldez-Cru, J. Nordström, and M. Vinyals. Using combinatorial benchmarks to probe the reasoning power of pseudo-boolean solvers. In *SAT*, pages 75–93, 2018.

[EGS12] S. Ermon, C. P. Gomes, and B. Selman. Uniform solution sampling using a constraint solver as an oracle. In *UAI*, pages 255–264, 2012.

[EN15] A. Erez and A. Nadel. Finding bounded path in graph using SMT for automatic clock routing. In *CAV*, pages 20–36, 2015.

[EN18] J. Elffers and J. Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *IJCAI*, pages 1291–1299, 2018.

[ES03] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[ES06] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[FBS19] K. Fazekas, A. Biere, and C. Scholl. Incremental inprocessing in SAT solving. In M. Janota and I. Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019.

[FM06] Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.

[FP01] A. M. Frisch and T. J. Peugniez. Solving non-boolean satisfiability problems with stochastic local search. In *IJCAI*, pages 282–290. Morgan Kaufmann, 2001.

[FS83] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. Computers*, 32(12):1137–1144, 1983.

[Gas77] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *International Joint Conference on Artificial Intelligence*, page 457, 1977.

[Gas79] J. Gaschnig. *Performance measurement and analysis of certain search algorithms.* PhD thesis, Carnegie-Mellon University, 1979.

[Gen02] I. P. Gent. Arc consistency in SAT. In *ECAI*, pages 121–125. IOS Press, 2002.

[Gen13] I. P. Gent. Optimal implementation of watched literals and more general techniques. *J. Artif. Intell. Res.*, 48:231–251, 2013.

[GeSSMS99] L. Guerra e Silva, L. M. Silveira, and J. Marques-Silva. Algorithms for solving boolean satisfiability in combinational circuits.

In *DATE*, pages 526–530. IEEE Computer Society / ACM, 1999.

[GIL18] É. Grégoire, Y. Izza, and J.-M. Lagniez. Boosting mcses enumeration. In *IJCAI*, pages 1309–1315, 2018.

[Gin93] M. L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res.*, 1:25–46, 1993.

[GK03] H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *LICS*, pages 55–64, 2003.

[GKKS12] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[GKSS08] C. P. Gomes, H. A. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 89–134. Elsevier, 2008.

[GL15] É. Grégoire and J.-M. Lagniez. On anti-subsumptive knowledge enforcement. In *LPAR*, volume 9450 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2015.

[GLM14a] É. Grégoire, J.-M. Lagniez, and B. Mazure. An experimentally efficient method for (mss, comss) partitioning. In *AAAI*, pages 2666–2673, 2014.

[GLM14b] É. Grégoire, J.-M. Lagniez, and B. Mazure. Multiple contraction through Partial-Max-SAT. In *ICTAI*, pages 321–327. IEEE Computer Society, 2014.

[GMP08] É. Grégoire, B. Mazure, and C. Piette. On approaches to explaining infeasibility of sets of boolean clauses. In *ICTAI*, pages 74–83, 2008.

[GN02] E. I. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002.

[GN03] E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, pages 10886–10891. IEEE Computer Society, 2003.

[GN04] I. P. Gent and P. Nightingale. A new encoding of alldifferent into SAT. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.

[GN08] E. Goldberg and Y. Novikov. Method and system for solving satisfiability problems, April 8 2008. US Patent 7,356,519.

[GS09] C. P. Gomes and A. Sabharwal. Exploiting runtime variation in complete solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 271–288. IOS Press, 2009.

[GSC97] C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *CP*, volume 1330 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1997.

[GSCK00] C. P. Gomes, B. Selman, N. Crato, and H. A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reasoning*, 24(1/2):67–100, 2000.

[GSK98] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, pages 431–437.

AAAI Press / The MIT Press, 1998.

[GSS09] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009.

[Heu18] M. J. H. Heule. Schur number five. In *AAAI*, pages 6598–6606, 2018.

[HHJKW17] M. J. H. Heule, W. A. Hunt Jr., M. Kaufmann, and N. Wetzler. Efficient, verified checking of propositional proofs. In *ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2017.

[HHJW13] M. J. H. Heule, W. A. Hunt Jr., and N. Wetzler. Trimming while checking clausal proofs. In *FMCAD*, pages 181–188. IEEE, 2013.

[HHJW14] M. J. H. Heule, W. A. Hunt Jr., and N. Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Softw. Test., Verif. Reliab.*, 24(8):593–607, 2014.

[HHW13] M. J. H. Heule, W. A. Hunt Jr., and N. Wetzler. Verifying refutations with extended resolution. In *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013.

[HJS18] M. J. H. Heule, M. Järvisalo, and M. Suda. Proceedings of SAT competition 2018. 2018.

[HK17] M. J. H. Heule and O. Kullmann. The science of brute force. *Commun. ACM*, 60(8):70–79, 2017.

[HKM16] M. J. H. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In N. Creignou and D. Le Berre, editors, *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

[HKSB17] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere. PRuning through satisfaction. In *Haifa Verification Conference*, volume 10629 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2017.

[HMPMS12] F. Heras, A. Morgado, J. Planes, and J. Marques-Silva. Iterative SAT solving for minimum satisfiability. In *ICTAI*, pages 922–927. IEEE Computer Society, 2012.

[HS15] M. J. H. Heule and T. Schaub. What's hot in the SAT and ASP competitions. In *AAAI*, pages 4322–4323, 2015.

[Hua07] J. Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, pages 2318–2323, 2007.

[Hua10] J. Huang. Extended clause learning. *Artif. Intell.*, 174(15):1277–1284, 2010.

[IJMS16] A. Ignatiev, M. Janota, and J. Marques-Silva. Quantified maximum satisfiability. *Constraints*, 21(2):277–302, 2016.

[IMM$^+$14] A. Ignatiev, A. Morgado, V. M. Manquinho, I. Lynce, and J. Marques-Silva. Progression in maximum satisfiability. In *ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 453–458. IOS Press, 2014.

[IMMS16] A. Ignatiev, A. Morgado, and J. Marques-Silva. Propositional abduction with implicit hitting sets. In *ECAI*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 1327–1335.

IOS Press, 2016.

[IMMS17a] A. Ignatiev, A. Morgado, and J. Marques-Silva. Cardinality encodings for graph optimization problems. In *IJCAI*, pages 652–658, 2017.

[IMMS17b] A. Ignatiev, A. Morgado, and J. Marques-Silva. On tackling the limits of resolution in SAT solving. In *SAT*, volume 10491 of *Lecture Notes in Computer Science*, pages 164–183. Springer, 2017.

[IMMS18] A. Ignatiev, A. Morgado, and J. Marques-Silva. PySAT: A python toolkit for prototyping with SAT oracles. In *SAT*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018.

[IMMV16] A. Ivrii, S. Malik, K. S. Meel, and M. Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1):41–58, 2016.

[IMPMS16] A. Ignatiev, A. Morgado, J. Planes, and J. Marques-Silva. Maximal falsifiability. *AI Commun.*, 29(2):351–370, 2016.

[IPMS15] A. Ignatiev, A. Previti, and J. Marques-Silva. SAT-based formula simplification. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2015.

[IPNaMS18] A. Ignatiev, F. Pereira, N. Narodytska, and J. ao Marques-Silva. A SAT-based approach to learn explainable decision sets. In *IJCAR*, volume 10900 of *Lecture Notes in Computer Science*, pages 627–645. Springer, 2018.

[Jac00] D. Jackson. Automating first-order relational logic. In *FSE*, pages 130–139, 2000.

[JaMS16] M. Janota and J. ao Marques-Silva. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell.*, 233:73–83, 2016.

[JBH10] M. Järvisalo, A. Biere, and M. J. H. Heule. Blocked clause elimination. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.

[JHB12] M. Järvisalo, M. J. H. Heule, and A. Biere. Inprocessing rules. In *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.

[JJ09] M. Järvisalo and T. A. Junttila. Limitations of restricted branching in clause learning. *Constraints*, 14(3):325–356, 2009.

[JKMSC12] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke. Solving QBF with counterexample guided refinement. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2012.

[JKMSC16] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016.

[JLMS15] M. Janota, I. Lynce, and J. Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Commun.*, 28(2):161–177, 2015.

[JLSS14] S. Jabbour, J. Lonlac, L. Sais, and Y. Salhi. Extending modern SAT solvers for models enumeration. In *IRI*, pages 803–810. IEEE

Computer Society, 2014.

[JMS11] M. Janota and J. Marques-Silva. Abstraction-based algorithm for 2QBF. In *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2011.

[JMS15] M. Janota and J. Marques-Silva. Solving QBF by clause selection. In *IJCAI*, pages 325–331. AAAI Press, 2015.

[JMSSS14] S. Jabbour, J. Marques-Silva, L. Sais, and Y. Salhi. Enumerating prime implicants of propositional formulae in conjunctive normal form. In *JELIA*, volume 8761 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2014.

[Jun04] U. Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, 2004.

[Kar18] M. Karpinski. Encoding cardinality constraints using standard encoding of generalized selection networks preserves arc-consistency. *Theor. Comput. Sci.*, 707:77–81, 2018.

[KGP01] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based boolean reasoning. pages 232–237, 2001.

[KKM94] R. Kohli, R. Krishnamurti, and P. Mirchandani. The minimum satisfiability problem. *SIAM J. Discrete Math.*, 7(2):275–283, 1994.

[KL14] B. Konev and A. Lisitsa. A SAT attack on the Erdős discrepancy conjecture. In *SAT*, volume 8561 of *Lecture Notes in Computer Science*, pages 219–226. Springer, 2014.

[KL15] B. Konev and A. Lisitsa. Computer-aided proof of Erdős discrepancy properties. *Artif. Intell.*, 224:103–118, 2015.

[KMS15] O. Kullmann and J. Marques-Silva. Computing maximal autarkies with few and simple oracle queries. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 138–155. Springer, 2015.

[Knu15] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability.* Addison-Wesley Professional, 1st edition, 2015.

[Kor08] K. Korovin. iprover - an instantiation-based theorem prover for first-order logic (system description). In *IJCAIR*, pages 292–298, 2008.

[Kro09] D. Kroening. Software verification. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 505–532. IOS Press, 2009.

[KRPH18] B. Kiesl, A. Rebola-Pardo, and M. J. H. Heule. Extended resolution simulates DRAT. In *IJCAR*, volume 10900 of *Lecture Notes in Computer Science*, pages 516–531. Springer, 2018.

[KS92] H. A. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[KS96] H. A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *AAAI*, pages 1194–1201, 1996.

[KS99] H. A. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *IJCAI*, pages 318–325. Morgan Kaufmann, 1999.

[KSTW05] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *AAAI*, pages 1368–1373. AAAI Press / The MIT Press, 2005.

[KV13] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *CAV*, pages 1–35, 2013.

[Lar89] T. Larrabee. Efficient generation of test patterns using boolean difference. In *ITC*, pages 795–802, 1989.

[Lar92] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[LB13] J.-M. Lagniez and A. Biere. Factoring out assumptions to speed up MUS extraction. In *SAT*, volume 7962 of *Lecture Notes in Computer Science*, pages 276–292. Springer, 2013.

[LBHHX14] K. Leyton-Brown, H. H. Hoos, F. Hutter, and L. Xu. Understanding the empirical hardness of *NP*-complete problems. *Commun. ACM*, 57(5):98–107, 2014.

[LBMS01] I. Lynce, L. Baptista, and J. Marques-Silva. Towards provably complete stochastic search algorithms for satisfiability. In *EPIA*, volume 2258 of *Lecture Notes in Computer Science*, pages 363–370. Springer, 2001.

[LBP10] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[Lev73] L. A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[LGPC16a] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *AAAI*, pages 3434–3440, 2016.

[LGPC16b] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In *SAT*, pages 123–140, 2016.

[LLBdLM18] J.-M. Lagniez, D. Le Berre, T. de Lima, and V. Montmirail. An assumption-based approach for solving the minimal S5-satisfiability problem. In *IJCAR*, pages 1–18, 2018.

[LLX$^+$17] M. Luo, C.-M. Li, F. Xiao, F. Manyà, and Z. Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *IJCAI*, pages 703–711, 2017.

[LM09] C. M. Li and F. Manyà. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 613–631. IOS Press, 2009.

[LMQZ10] C. M. Li, F. Manyà, Z. Quan, and Z. Zhu. Exact MinSAT solving. In *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 363–368. Springer, 2010.

[LMS02] I. Lynce and J. Marques-Silva. Building state-of-the-art SAT solvers. In *ECAI*, pages 166–170. IOS Press, 2002.

[LMS03] I. Lynce and J. Marques-Silva. An overview of backtrack search satisfiability algorithms. *Ann. Math. Artif. Intell.*, 37(3):307–326, 2003.

[LOM$^+$18] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh. Machine learning-based restart policy for CDCL SAT solvers. In *SAT*, pages 94–110, 2018.

[LPMMS16] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2):223–250, 2016.

[LS08] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

[LSZ93] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.

[MB19] S. Möhle and A. Biere. Backing backtracking. In *SAT*, pages 250–266, 2019.

[McC56] E. J. McCluskey. Minimization of Boolean functions. *Bell system technical Journal*, 35(6):1417–1444, 1956.

[McM03] K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[McM18] K. L. McMillan. Interpolation and model checking. In *Handbook of Model Checking*, pages 421–446. Springer, 2018.

[MDMS14] A. Morgado, C. Dodaro, and J. Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In *CP*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014.

[MHB12] N. Manthey, M. J. H. Heule, and A. Biere. Automated reencoding of boolean formulas. In *Haifa Verification Conference*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2012.

[MHL$^+$13] A. Morgado, F. Heras, M. H. Liffiton, J. Planes, and J. Marques-Silva. Iterative and core-guided MaxSA solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.

[MIPMS16] C. Mencía, A. Ignatiev, A. Previti, and J. Marques-Silva. MCS extraction with sublinear oracle queries. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 342–360. Springer, 2016.

[Mit05] D. G. Mitchell. A SAT solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.

[MJML14] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce. Incremental cardinality constraints for maxsat. In *CP*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014.

[MMM08] M. Moskewicz, C. Madigan, and S. Malik. Method and system for efficient implementation of boolean satisfiability. Available from `https://patents.google.com/patent/US7418369B2/en`, August 26 2008. US Patent 7,418,369.

[MMSP09] V. M. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for weighted boolean optimization. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 495–508. Springer, 2009.

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535.

ACM, 2001.

[MPMS15] C. Mencía, A. Previti, and J. Marques-Silva. Literal-based MCS extraction. In *IJCAI*, pages 1973–1979. AAAI Press, 2015.

[MPS14] N. Manthey, T. Philipp, and P. Steinke. A more compact translation of pseudo-boolean constraints into CNF such that generalized arc consistency is maintained. In *KI*, volume 8736 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 2014.

[MS95] J. P. Marques-Silva. *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits.* PhD thesis, University of Michigan, May 1995.

[MS98] J. Marques-Silva. An overview of backtrack search satisfiability algorithms. In *ISAIM*, 1998.

[MS99] J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.

[MS08] J. Marques-Silva. Practical applications of boolean satisfiability. In *WODES*, pages 74–80. IEEE, 2008.

[MS10] J. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL*, pages 9–14. IEEE Computer Society, 2010.

[MSGeS03] J. Marques-Silva and L. Guerra e Silva. Solving satisfiability in combinational circuits. *IEEE Design & Test of Computers*, 20(4):16–21, 2003.

[MSHJ$^+$13] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In *IJCAI*, pages 615–622. IJCAI/AAAI, 2013.

[MSIM$^+$14] J. Marques-Silva, A. Ignatiev, A. Morgado, V. M. Manquinho, and I. Lynce. Efficient autarkies. In *ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 603–608. IOS Press, 2014.

[MSJB13] J. Marques-Silva, M. Janota, and A. Belov. Minimal sets over monotone predicates in boolean formulae. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 592–607. Springer, 2013.

[MSJIM15] J. Marques-Silva, M. Janota, A. Ignatiev, and A. Morgado. Efficient model based diagnosis with maximum satisfiability. In *IJCAI*, pages 1966–1972. AAAI Press, 2015.

[MSJL10] J. Marques-Silva, M. Janota, and I. Lynce. On computing backbones of propositional theories. In *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 15–20. IOS Press, 2010.

[MSJM17] J. Marques-Silva, M. Janota, and C. Mencía. Minimal sets on propositional formulae. problems and reductions. *Artif. Intell.*, 252:22–50, 2017.

[MSKC14] A. Metodi, R. Stern, M. Kalech, and M. Codish. A novel sat-based approach to model based diagnosis. *J. Artif. Intell. Res.*, 51:377–411, 2014.

[MSL11] J. Marques-Silva and I. Lynce. On improving MUS extraction algorithms. In *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2011.

[MSLM09] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[MSM18] J. Marques-Silva and S. Malik. Propositional SAT solving. In *Handbook of Model Checking*, pages 247–275. Springer, 2018.

[MSP07] J. Marques-Silva and J. Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, abs/0712.1097, 2007.

[MSS$^+$91] P. C. McGeer, A. Saldanha, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Timing analysis and delay-fault test generation using path-recursive functions. In *ICCAD*, pages 180–183. IEEE Computer Society, 1991.

[MSS93] J. Marques-Silva and K. A. Sakallah. Space pruning heuristics for path sensitization in test pattern generation. Technical Report CSE-TR-178-93, University of Michigan, 1993.

[MSS94] J. Marques-Silva and K. A. Sakallah. Dynamic search-space pruning techniques in path sensitization. In *DAC*, pages 705–711. ACM Press, 1994.

[MSS96a] J. Marques-Silva and K. A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. Technical Report RT-04-96, INESC, May 1996.

[MSS96b] J. Marques-Silva and K. A. Sakallah. Grasp – a new search algorithm for satisfiability. Technical Report CSE-TR-292-96, University of Michigan, April 1996.

[MSS96c] J. Marques-Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[MSS99] J. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[MSS00] J. Marques-Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC*, pages 675–680. ACM, 2000.

[MZ09] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[MZK$^+$99] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133, 1999.

[Nad10] A. Nadel. Boosting minimal unsatisfiable core extraction. In *FMCAD*, pages 221–229, 2010.

[NBE12] A. Nöhrer, A. Biere, and A. Egyed. Managing SAT inconsistencies with HUMUS. In *VaMoS*, pages 83–91. ACM, 2012.

[NBMS18] N. Narodytska, N. Bjørner, M.-C. Marinescu, and M. Sagiv. Core-guided minimal correction set and core enumeration. In *IJCAI*, pages 1353–1361, 2018.

[NIPMS18] N. Narodytska, A. Ignatiev, F. Pereira, and J. Marques-Silva. Learning optimal decision trees with SAT. In *IJCAI*, pages 1362–

1368, 2018.

[NR12] A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In *SAT*, pages 242–255, 2012.

[NR18] A. Nadel and V. Ryvchin. Chronological backtracking. In *SAT*, pages 111–121, 2018.

[NRS13] A. Nadel, V. Ryvchin, and O. Strichman. Efficient MUS extraction with resolution. In *FMCAD*, pages 197–200, 2013.

[NRS14] A. Nadel, V. Ryvchin, and O. Strichman. Ultimately incremental SAT. In *SAT*, pages 206–218, 2014.

[Oh15] C. Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In *SAT*, pages 307–323, 2015.

[OLH$^+$13] T. Ogawa, Y. Liu, R. Hasegawa, M. Koshimura, and H. Fujita. Modulo based CNF encoding of cardinality constraints and its application to MaxSAT solvers. In *ICTAI*, pages 9–17. IEEE Computer Society, 2013.

[OSC09] O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[PD07] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[PD09] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers with restarts. In *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 654–668. Springer, 2009.

[PD11] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.

[PIMMS15] A. Previti, A. Ignatiev, A. Morgado, and J. Marques-Silva. Prime compilation of non-clausal formulae. In *IJCAI*, pages 1980–1988. AAAI Press, 2015.

[PMJaMS18] A. Previti, C. Mencía, M. Järvisalo, and J. ao Marques-Silva. Premise set caching for enumerating minimal correction subsets. In *AAAI*, pages 6633–6640. AAAI Press, 2018.

[Pre07] S. D. Prestwich. Variable dependency in local search: Prevention is better than cure. In *SAT*, pages 107–120, 2007.

[Pre09] S. D. Prestwich. CNF encodings. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 75–97. IOS Press, 2009.

[Pre15] A. Previti. *On the Duality Relating Sets.* PhD thesis, University College Dublin, 2015.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Qui52] W. V. Quine. The problem of simplifying truth functions. *American mathematical monthly*, pages 521–531, 1952.

[Qui55] W. V. Quine. A way to simplify truth functions. *American mathematical monthly*, pages 627–631, 1955.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[Rin09] J. Rintanen. Planning and SAT. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 483–504. IOS Press, 2009.

[RM09] O. Roussel and V. M. Manquinho. Pseudo-boolean and cardinality constraints. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 695–733. IOS Press, 2009.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[RS16] M. N. Rabe and S. A. Seshia. Incremental determinization. In *SAT*, pages 375–392, 2016.

[RT15] M. N. Rabe and L. Tentrup. CAQE: A certifying QBF solver. In *FMCAD*, pages 136–143, 2015.

[Rya04] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, February 2004.

[SA89] M. H. Schulz and E. Auth. Improved deterministic test pattern generation with applications to redundancy identification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(7):811–816, 1989.

[Sak09] K. A. Sakallah. Symmetry and satisfiability. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press, 2009.

[SB09] N. Sörensson and A. Biere. Minimizing learned clauses. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[SBJ16] P. Saikko, J. Berg, and M. Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In *SAT*, pages 539–546, 2016.

[SBK05] T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2005.

[SBSV96] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.

[SBTLB17] T. Soh, M. Banbara, N. Tamura, and D. Le Berre. Solving multiobjective discrete optimization problems with propositional minimal model generation. In *CP*, volume 10416 of *Lecture Notes in Computer Science*, pages 596–614. Springer, 2017.

[SI09] C. Sinz and M. Iser. Problem-sensitive restart heuristics for the DPLL procedure. In *SAT*, pages 356–362, 2009.

[SI10] T. Soh and K. Inoue. Identifying necessary reactions in metabolic pathways by minimal model generation. In *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 277–282. IOS Press, 2010.

[Sin05] C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *CP*, pages 827–831, 2005.

[SLM92] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, pages 440–446, 1992.

[SNC09] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[SP04] S. Subbarayan and D. K. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *SAT*, 2004.

[SS77] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[SSS12] A. Sabharwal, H. Samulowitz, and M. Sellmann. Learning back-clauses in SAT. In *SAT*, pages 498–499, 2012.

[Stu13] P. J. Stuckey. There are no CNF problems. In *SAT*, pages 19–21, 2013.

[SWJ16] P. Saikko, J. P. Wallner, and M. Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In *KR*, pages 104–113, 2016.

[SZGN17] X. Si, X. Zhang, R. Grigore, and M. Naik. Maximum satisfiability in software analysis: Applications and techniques. In *CAV*, pages 68–94, 2017.

[SZO$^+$18] A. A. Semenov, O. Zaikin, I. V. Otpuschennikov, S. Kochemazov, and A. Ignatiev. On cryptographic attacks using backdoors for SAT. In *AAAI*, pages 6641–6648. AAAI Press, 2018.

[TBW04] C. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with DPLL search. In *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer, 2004.

[TJ07] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647, 2007.

[Van08] A. Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.

[Van12] A. Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.

[VG02] A. Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *ISAIM*, 2002.

[VG09] A. Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *SAT*, pages 141–146, 2009.

[VG11] A. Van Gelder. Generalized conflict-clause strengthening for satisfiability solvers. In *SAT*, pages 329–342, 2011.

[Vor14] A. Voronkov. AVATAR: the architecture for first-order theorem provers. In *CAV*, pages 696–710, 2014.

[VWM15] Y. Vizel, G. Weissenbacher, and S. Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.

[WA09] T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Applied Logic*, 7(1):26–40, 2009.

[Wal00] T. Walsh. SAT v CSP. In *CP*, volume 1894 of *Lecture Notes in*

Computer Science, pages 441–456. Springer, 2000.

[War98] J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Inf. Process. Lett.*, 68(2):63–69, 1998.

[Web06] T. Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. In *IWIL*, pages 44–62, 2006.

[WGS03] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI*, pages 1173–1178. Morgan Kaufmann, 2003.

[WHHJ13] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. Mechanical verification of SAT refutations with extended resolution. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2013.

[WHHJ14] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.

[WvM98] J. P. Warners and H. van Maaren. A two-phase algorithm for solving a class of hard satisfiability problems. *Oper. Res. Lett.*, 23(3-5):81–88, 1998.

[XHHLB08] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.

[XLE$^+$16] Y. Xue, Z. Li, S. Ermon, C. P. Gomes, and B. Selman. Solving marginal MAP problems with NP oracles and parity constraints. In *NIPS*, pages 1127–1135, 2016.

[Zha97] H. Zhang. SATO: an efficient propositional prover. In *CADE*, pages 272–275, 1997.

[ZLMA12] Z. Zhu, C. M. Li, F. Manyà, and J. Argelich. A new encoding from MinSAT into MaxSAT. In *CP*, volume 7514 of *Lecture Notes in Computer Science*, pages 455–463. Springer, 2012.

[ZM88] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *AAAI*, pages 155–160, 1988.

[ZM02] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2002.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885. IEEE Computer Society, 2003.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285. IEEE Computer Society, 2001.

[ZS00] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *J. Autom. Reasoning*, 24(1/2):277–296, 2000.

[ZWM11] C. S. Zhu, G. Weissenbacher, and S. Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In *FMCAD*, pages 63–66. FMCAD Inc., 2011.

[ZWSM11] C. S. Zhu, G. Weissenbacher, D. Sethi, and S. Malik. SAT-based techniques for determining backbones for post-silicon fault localisation. In *HLDVT*, pages 84–91. IEEE Computer Society, 2011.