Handbook of Satisfiability
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsch
IOS Press, 2008

Chapter 4

Conflict-Driven Clause Learning SAT Solvers

Joao Marques-Silva, Ines Lynce and Sharad Malik

4.1. Introduction

One of the main reasons for the widespread use of SAT in many applications is that *Conflict-Driven Clause Learning* (CDCL) Boolean Satisfiability (SAT) solvers are so effective in practice. Since their inception in the mid-90s, CDCL SAT solvers have been applied, in many cases with remarkable success, to a number of practical applications. Examples of applications include hardware and software model checking, planning, equivalence checking, bioinformatics, hardware and software test pattern generation, software package dependencies, and cryptography. This chapter surveys the organization of CDCL solvers, from the original solvers that inspired modern CDCL SAT solvers, to the most recent and proven techniques.

The organization of CDCL SAT solvers is primarily inspired by DPLL solvers. As a result, and even though the chapter is self-contained, a reasonable knowledge of the organization of DPLL is assumed. In order to offer a detailed account of CDCL SAT solvers, a number of concepts have to be introduced, which serve to formalize the operations implemented by any DPLL SAT solver.

DPLL corresponds to backtrack search, where at each step a variable and a propositional value are selected for branching purposes. With each branching step, two values can be assigned to a variable, either 0 or 1. Branching corresponds to assigning the chosen value to the chosen variable. Afterwards, the logical consequences of each branching step are evaluated. Each time an unsatisfied clause (i.e. a *conflict*) is identified, *backtracking* is executed. Backtracking corresponds to undoing branching steps until an unflipped branch is reached. When both values have been assigned to the selected variable at a branching step, backtracking will undo this branching step. If for the first branching step both values have been considered, and backtracking undoes this first branching step, then the CNF formula can be declared *unsatisfiable*. This kind of backtracking is called *chronological backtracking*. An alternative backtracking scheme is *non-chronological backtracking*, which is described later in this chapter. A more detailed description of the DPLL algorithm is given in Part 1, Chapter 3.

Besides using DPLL, building a state-of-the-art CDCL SAT solver involves a number of additional key techniques:

- Learning new clauses from conflicts during backtrack search [MSS96].
- Exploiting structure of conflicts during clause learning [MSS96].
- Using *lazy data structures* for the representation of formulas [MMZ+01].
- Branching heuristics must have low computational overhead, and must receive feedback from backtrack search [MMZ+01].
- Periodically restarting backtrack search [GSK98].
- Additional techniques include deletion policies for learnt clauses [GN02], the actual implementation of lazy data structures [Rya04], the organization of unit propagation [LSB05], among others.

The chapter is organized as follows. The next section introduces the notation used throughout the chapter. Afterwards, Section 4.3 summarizes the organization of modern CDCL SAT solvers. Section 4.4 details conflict analysis, the procedure used for learning new clauses. Section 4.5 outlines more recent techniques that have been shown to be effective in practice. The chapter concludes in Section 4.6 by providing a historical perspective of the work on CDCL SAT solvers.

4.2. Notation

In this chapter, propositional formulas are represented in Conjunctive Normal Form (CNF). A finite set of Boolean variables is assumed $X = \{x_1, x_2, x_3, \ldots, x_n\}$. A CNF formula φ consists of a conjunction of clauses ω, each of which consists of a disjunction of literals. A literal is either a variable x_i or its complement $\neg x_i$. A CNF formula can also be viewed as a set of clauses, and each clause can be viewed as a set of literals. Throughout this chapter, the representation used will be clear from the context.

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

$$\varphi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \tag{4.1}$$

The alternative set representation is:

$$\varphi = \{\{x_1, \neg x_2\}, \{x_2, x_3\}, \{x_2, \neg x_4\}, \{\neg x_1, \neg x_3, x_4\}\} \tag{4.2}$$

In the context of search algorithms for SAT, variables can be *assigned* a logic value, either 0 or 1. Alternatively, variables may also be *unassigned*. Assignments to the problem variables can be defined as a function $\nu : X \to \{0, u, 1\}$, where u denotes an *undefined* value used when a variable has not been assigned a value in $\{0, 1\}$. 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*.

Assignments serve for computing the values of literals, clauses and the complete CNF formula, respectively, l^ν, ω^ν and φ^ν. A total order is defined on the possible assignments, $0 < u < 1$. Moreover, $1 - u = u$. As a result, the following

definitions apply:

$$l^\nu = \begin{cases} \nu(x_i) & \text{if } l = x_i \\ 1 - \nu(x_i) & \text{if } l = \neg x_i \end{cases} \tag{4.3}$$

$$\omega^\nu = \max\{l^\nu \,|\, l \in \omega\} \tag{4.4}$$

$$\varphi^\nu = \min\{\omega^\nu \,|\, \omega \in \varphi\} \tag{4.5}$$

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.

Clauses are characterized as *unsatisfied*, *satisfied*, *unit* or *unresolved*. A clause is unsatisfied 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 DPLL, logical consequences are derived with unit propagation. Unit propagation is applied after each branching step (and also during preprocessing), 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 backtracks.

In CDCL SAT solvers, each variable x_i is characterized by a number of properties, including the *value*, the *antecedent* and the *decision level*, denoted respectively by $\nu(v_i) \in \{0, u, 1\}$, $\alpha(x_i) \in \varphi \cup \{\text{NIL}\}$, and $\delta(x_i) \in \{-1, 0, 1, \ldots, |X|\}$. 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 ω used for implying variable x_i is said to be the antecedent of x_i, $\alpha(x_i) = \omega$. For variables that are decision variables or are unassigned, the antecedent is NIL. Hence, antecedents are only defined for variables whose value is implied by other assignments. The decision level of a variable x_i 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 -1, $\delta(x_i) = -1$. The decision level associated with variables used for branching steps (i.e. *decision assignments*) is specified by the search process, and denotes the current depth of the *decision stack*. Hence, a variable x_i associated with a decision assignment is characterized by having $\alpha(x_i) = \text{NIL}$ and $\delta(x_i) > 0$. More formally, the decision level of x_i with antecedent ω is given by:

$$\delta(x_i) = \max(\{0\} \cup \{\delta(x_j) \,|\, x_j \in \omega \land x_j \neq x_i\}) \tag{4.6}$$

i.e. the decision level of an implied literal is either the highest decision level of the implied literals in a unit clause, or it is 0 in case the clause is unit. The notation $x_i = v \,@\, d$ is used to denote that $\nu(x_i) = v$ and $\delta(x_i) = d$. Moreover, the decision level of a literal is defined as the decision level of its variable, $\delta(l) = \delta(x_i)$ if $l = x_i$ or $l = \neg x_i$.

Example 4.2.2 (Decision Levels & Antecedents)**.** Consider the CNF formula:

$$\begin{aligned}\varphi &= \omega_1 \land \omega_2 \land \omega_3 \\ &= (x_1 \lor \neg x_4) \land (x_1 \lor x_3) \land (\neg x_3 \lor x_2 \lor x_4)\end{aligned} \tag{4.7}$$

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, $\alpha(x_3) = \omega_2$ and $\alpha(x_2) = \omega_3$.

During the execution of a DPLL-style SAT solver, assigned variables as well as their antecedents define a directed acyclic graph $I = (V_I, E_I)$, referred to as the *implication graph* [MSS96].

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

The edges of I are formally defined below. Let $z, z_1, z_2 \in V_I$ be vertices in I (observe that the vertices can be variables or the special vertex κ). In order to derive the conditions for existence of edges in I, a number of predicates needs to be defined first. Predicate $\lambda(z, \omega)$ takes value 1 iff ω has a literal in z, and is defined as follows:

$$\lambda(z, \omega) = \begin{cases} 1 & \text{if } z \in \omega \lor \neg z \in \omega \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

This predicate can now be used for testing the value of a literal in z in a given clause. Predicate $\nu_0(z, \omega)$ takes value 1 iff z is a literal in ω and the value of the literal is 0:

$$\nu_0(z, \omega) = \begin{cases} 1 & \text{if } \lambda(z, \omega) \land z \in \omega \land \nu(z) = 0 \\ 1 & \text{if } \lambda(z, \omega) \land \neg z \in \omega \land \nu(z) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

Predicate $\nu_1(z, \omega)$ takes value 1 iff z is a literal in ω and the value of the literal is 1:

$$\nu_1(z, \omega) = \begin{cases} 1 & \text{if } \lambda(z, \omega) \land z \in \omega \land \nu(z) = 1 \\ 1 & \text{if } \lambda(z, \omega) \land \neg z \in \omega \land \nu(z) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

As a result, there is an edge from z_1 to z_2 in I iff the following predicate takes value 1:

$$\epsilon(z_1, z_2) = \begin{cases} 1 & \text{if } z_2 = \kappa \land \lambda(z_1, \alpha(\kappa)) \\ 1 & \text{if } z_2 \neq \kappa \land \alpha(z_2) = \omega \land \nu_0(z_1, \omega) \land \nu_1(z_2, \omega) \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

Consequently, the set of edges E_I of the implication graph I is given by:

$$E_I = \{(z_1, z_2) \,|\, \epsilon(z_1, z_2) = 1\} \tag{4.12}$$

Finally, observe that a labeling function for associating a clause with each edge can also be defined. Let $\iota : V_I \times V_I \to \varphi$ be the labeling function. Then $\iota(z_1, z_2)$, with $z_1, z_2 \in V_I$ and $(z_1, z_2) \in E_I$, is defined by $\iota(z_1, z_2) = \alpha(z_2)$.

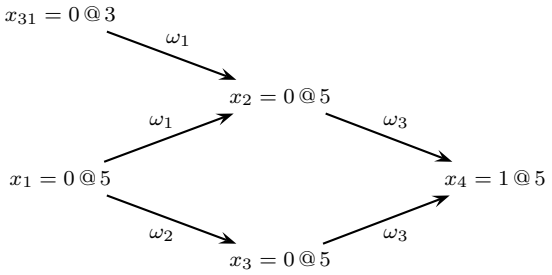


Figure 4.1. Implication graph for example 4.2.3

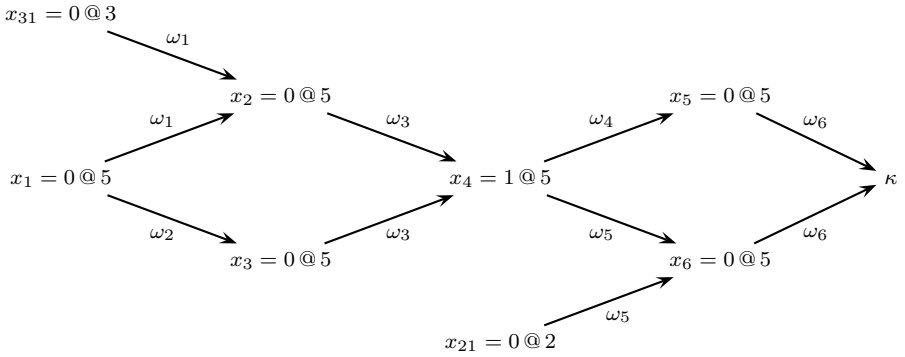


Figure 4.2. Implication graph for example 4.2.4

Example 4.2.3 (Implication Graph without Conflict)**.** Consider the CNF formula:

$$\begin{aligned} \varphi_1 &= \omega_1 \wedge \omega_2 \wedge \omega_3 \\ &= (x_1 \vee x_{31} \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \end{aligned} \tag{4.13}$$

Assume decision assignment $x_{31} = 0@3$. Moreover, assume that the current decision assignment is $x_1 = 0@5$. The resulting implication graph is shown in figure 4.1.

Example 4.2.4 (Implication Graph with Conflict)**.** Consider the CNF formula:

$$\begin{aligned} \varphi_1 &= \omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 \wedge \omega_5 \wedge \omega_6 \\ &= (x_1 \vee x_{31} \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge \\ &\quad (\neg x_4 \vee \neg x_5) \wedge (x_{21} \vee \neg x_4 \vee \neg x_6) \wedge (x_5 \vee x_6) \end{aligned} \tag{4.14}$$

Assume decision assignments $x_{21} = 0@2$ and $x_{31} = 0@3$. Moreover, assume the current decision assignment $x_1 = 0@5$. The resulting implication graph is shown in figure 4.2, and yields a conflict because clause $(x_5 \vee x_6)$ becomes unsatisfied.

4.3. Organization of CDCL Solvers

Algorithm 1 shows the standard organization of a CDCL SAT solver, which essentially follows the organization of DPLL. With respect to DPLL, the main

Algorithm 1 Typical CDCL algorithm

```
CDCL(φ, ν)
 1  if (UnitPropagation(φ, ν) == CONFLICT)
 2     then return UNSAT
 3  dl ← 0                                          ▷ Decision level
 4  while (not AllVariablesAssigned(φ, ν))
 5        do (x, v) = PickBranchingVariable(φ, ν)                    ▷ Decide stage
 6           dl ← dl + 1                          ▷ Increment decision level due to new decision
 7           ν ← ν ∪ {(x, v)}
 8           if (UnitPropagation(φ, ν) == CONFLICT)                  ▷ Deduce stage
 9              then β = ConflictAnalysis(φ, ν)                      ▷ Diagnose stage
10                   if (β < 0)
11                      then return UNSAT
12                      else Backtrack(φ, ν, β)
13                           dl ← β              ▷ Decrement decision level due to backtracking
14  return SAT
```

differences are the call to function ConflictAnalysis each time a conflict is identified, and the call to Backtrack when backtracking takes place. Moreover, the Backtrack procedure allows for backtracking non-chronologically.

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

- UnitPropagation consists of the iterated application of the unit clause rule. If an unsatisfied clause is identified, then a conflict indication is returned.
- PickBranchingVariable consists of selecting a variable to assign and the respective value.
- ConflictAnalysis consists of analyzing the most recent conflict and learning a new clause from the conflict. The organization of this procedure is described in section 4.4.
- Backtrack backtracks to the decision level computed by ConflictAnalysis.
- AllVariablesAssigned tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable. An alternative criterion to stop execution of the algorithm is to check whether all clauses are satisfied. However, in modern SAT solvers that use lazy data structures, clause state cannot be maintained accurately, and so the termination criterion must be whether all variables are assigned.

Arguments to the auxiliary functions are assumed to be passed by reference. Hence, φ and ν are supposed to be modified during execution of the auxiliary functions.

The typical CDCL algorithm shown does not account for a few often used techniques, namely search restarts [GSK98, BMS00] and implementation of clause deletion policies [GN02]. Search restarts cause the algorithm to restart itself, but already learnt clauses are kept. Clause deletion policies are used to decide learnt clauses that can be deleted. Clause deletion allows the memory usage of the SAT

solver to be kept under control.

4.4. Conflict Analysis

This section outlines the conflict analysis procedure used by modern SAT solvers.

4.4.1. Learning Clauses from Conflicts

Each time the CDCL SAT solver identifies a conflict due to unit propagation, the ConflictAnalysis procedure is invoked. As a result, one or more new clauses are learnt, and a backtracking decision level is computed. The conflict analysis procedure analyzes the structure of unit propagation and decides which literals to include in the learnt clause.

The decision levels associated with assigned variables define a partial order of the variables. Starting from a given unsatisfied clause (represented in the implication graph with vertex κ), the conflict analysis procedure visits variables implied at the most recent decision level (i.e. the current largest decision level), identifies the antecedents of visited variables, and keeps from the antecedents the literals assigned at decision levels *less* than the most recent decision level. This process is repeated until the most recent decision variable is visited.

Let d be the current decision level, let x_i be the decision variable, let $\nu(x_i) = v$ be the decision assignment, and let ω_j be an unsatisfied clause identified with unit propagation. In terms of the implication graph, the conflict vertex κ is such that $\alpha(\kappa) = \omega_j$. Moreover, let $\odot$ represent the resolution operator. For two clauses ω_j and ω_k, for which there is a unique variable x such that one clause has a literal x and the other has literal $\neg x$, $\omega_j \odot \omega_k$ contains all the literals of ω_j and ω_k with the exception of x and $\neg x$.

The clause learning procedure used in SAT solvers can be defined by a sequence of selective resolution operations [MSS00, BKS04], that at each step yields a new temporary clause. First, define a predicate that holds if a clause ω has an implied literal l assigned at the current decision level d:

$$\xi(\omega, l, d) = \begin{cases} 1 & \text{if } l \in \omega \wedge \delta(l) = d \wedge \alpha(l) \neq \text{NIL} \\ 0 & \text{otherwise} \end{cases} \tag{4.15}$$

Let $\omega_L^{d,i}$, with $i = 0, 1, ...$, be the intermediate clause obtained after i resolution operations. Using the predicate defined by (4.15), this intermediate clause can be defined as follows:

$$\omega_L^{d,i} = \begin{cases} \alpha(\kappa) & \text{if } i = 0 \\ \omega_L^{d,i-1} \odot \alpha(l) & \text{if } i \neq 0 \wedge \xi(\omega_L^{d,i-1}, l, d) = 1 \\ \omega_L^{d,i-1} & \text{if } i \neq 0 \wedge \forall_l \, \xi(\omega_L^{d,i-1}, l, d) = 0 \end{cases} \tag{4.16}$$

Equation (4.16) can be used for formalizing the clause learning procedure. The first condition, $i = 0$, denotes the initialization step given κ in I, where all literals in the unsatisfied clause are added to the first intermediate clause clause. Afterwards, at each step i, a literal l assigned at the current decision level d is

Table 4.1. Resolution steps during clause learning

$\omega_L^{5,0} = \{x_5, x_6\}$	Literals in $\alpha(\kappa)$
$\omega_L^{5,1} = \{\neg x_4, x_6\}$	Resolve with $\alpha(x_5) = \omega_4$
$\omega_L^{5,2} = \{\neg x_4, x_{21}\}$	Resolve with $\alpha(x_6) = \omega_5$
$\omega_L^{5,3} = \{x_2, x_3, x_{21}\}$	Resolve with $\alpha(x_4) = \omega_3$
$\omega_L^{5,4} = \{x_1, x_{31}, x_3, x_{21}\}$	Resolve with $\alpha(x_2) = \omega_1$
$\omega_L^{5,5} = \{x_1, x_{31}, x_{21}\}$	Resolve with $\alpha(x_3) = \omega_2$
$\omega_L^{5,6} = \{x_1, x_{31}, x_{21}\}$	No more resolution operations given (4.16)

selected and the intermediate clause (i.e. $\omega_L^{d,i-1}$) is resolved with the antecedent of l.

For an iteration i such that $\omega_L^{d,i} = \omega_L^{d,i-1}$, then a *fixed point* is reached, and $\omega_L \triangleq \omega_L^{d,i}$ represents the new learnt clause. Observe that the number of resolution operations represented by (4.16) is no greater than $|X|$.

Modern SAT solvers implement an additional refinement of equation (4.16), by further exploiting the structure of implied assignments. This is discussed in sub-section 4.4.3.

Example 4.4.1 (Clause Learning)**.** Consider example 4.2.4. Applying clause learning to this example, results in the intermediate clauses shown in table 4.1. The resulting learnt clause is $(x_1 \vee x_{31} \vee x_{21})$. Alternatively, this clause can be obtained by inspecting the graph in figure 4.2, and selecting the literals assigned at decision levels less than the current decision level 5 (i.e. $x_{31} = 0@3$ and $x_{21} = 0@2$), and by selecting the corresponding decision assignment (i.e. $x_1 = 0@5$).

4.4.2. Completeness Issues

DPLL is a sound and complete algorithm for SAT [DP60, DLL62]. CDCL SAT solvers implement DPLL, but can learn new clauses and backtrack non-chronologically.

Clause learning with conflict analysis does not affect soundness or completeness. Conflict analysis identifies new clauses using the resolution operation. Hence each learnt clause can be inferred from the original clauses and other learnt clauses by a sequence of resolution steps. If ω_L is the new learnt clause, then φ is satisfiable if and only if $\varphi \cup \{\omega_L\}$ is also satisfiable. Moreover, the modified backtracking step also does not affect soundness or completeness, since backtracking information is obtained from each new learnt clause. Proofs of soundness and completeness for different variations of CDCL SAT solvers can be found in [MS95, MSS99, Zha03].

4.4.3. Exploiting Structure with UIPs

As can be concluded from equation (4.16), the structure of implied assignments induced by unit propagation is a key aspect of the clause learning procedure [MSS96].

This is one of the most relevant aspects of clause learning in SAT solvers. Moreover, the idea of exploiting the structure induced by unit propagation was further exploited with *Unit Implication Points* (UIPs) [MSS96]. A UIP is a dominator [1] in the implication graph, and represents an alternative decision assignment at the current decision level that results in the same conflict. The main motivation for identifying UIPs is to reduce the size of learnt clauses.

There are sophisticated algorithms for computing dominators in directed acyclic graphs [Tar74]. For the case of the implication graph, UIPs can be identified in linear time [MSS94], and so do not add significant overhead to the clause learning procedure.

In the implication graph, there is a UIP at decision level d, when the number of literals in $\omega_L^{d,i}$ assigned at decision level d is 1. Let $\sigma(\omega, d)$ be the number of literals in ω assigned at decision level d. $\sigma(\omega, d)$ can be defined as follows:

$$\sigma(\omega, d) = |\{l \in \omega \,|\, \delta(l) = d\}| \tag{4.17}$$

As a result, the clause learning procedure with UIPs is given by:

$$\omega_L^{d,i} = \begin{cases} \alpha(\kappa) & \text{if } i = 0 \\ \omega_L^{d,i-1} \odot \alpha(l) & \text{if } i \neq 0 \wedge \xi(\omega_L^{d,i-1}, l, d) = 1 \\ \omega_L^{d,i-1} & \text{if } i \neq 0 \wedge \sigma(\omega_L^{d,i-1}, d) = 1 \end{cases} \tag{4.18}$$

Equation (4.18) allows creating a clause containing literals from the learnt clause until the first UIP is identified. It is simple to develop equations for learning clauses for each additional UIP. However, as explained below, this is unnecessary in practice, since the most effective CDCL SAT solvers stop clause learning at the first UIP [ZMMM01]. Moreover, clause learning could potentially stop at any UIP, being quite straightforward to conclude that the set of literals of a clause learnt at the first UIP has clear advantages. Considering the largest decision level of the literals of the clause learnt at each UIP, the clause learnt at the first UIP is guaranteed to contain the smallest one. This guarantees the highest backtrack jump in the search tree.

Example 4.4.2 (Clause Learning with UIPs)**.** Consider again example 4.2.4. The default clause learning procedure would learn clause $(x_1 \vee x_{31} \vee x_{21})$ (see example 4.4.1). However, by taking into consideration that $x_4 = 1@5$ is a UIP, applying clause learning yields the resulting learnt clause $(\neg x_4 \vee x_{21})$, for which the intermediate clauses are shown in table 4.2. One advantage of this new clause is that it has a smaller size than the clause learnt in example 4.4.1.

4.4.4. Backtracking Schemes

Although the clause learning scheme of CDCL SAT solvers has remained essentially unchanged since it was proposed in GRASP [MSS96, MSS99], the actual backtracking step has been further refined. Motivated by the inexpensive backtracking achieved with lazy data structures, the authors of Chaff proposed to

[1] A vertex u dominates another vertex x in a directed graph if every path from x to another vertex κ contains u [Tar74]. In the implication graph, a UIP u dominates the decision vertex x with respect to the conflict vertex κ.

Table 4.2. Resolution steps during clause learning with UIPs

$\omega_L^{5,0} = \{x_5, x_6\}$	Literals in $\alpha(\kappa)$
$\omega_L^{5,1} = \{\neg x_4, x_6\}$	Resolve with $\alpha(x_5) = \omega_4$
$\omega_L^{5,2} = \{\neg x_4, x_{21}\}$	No more resolution operations given (4.18)

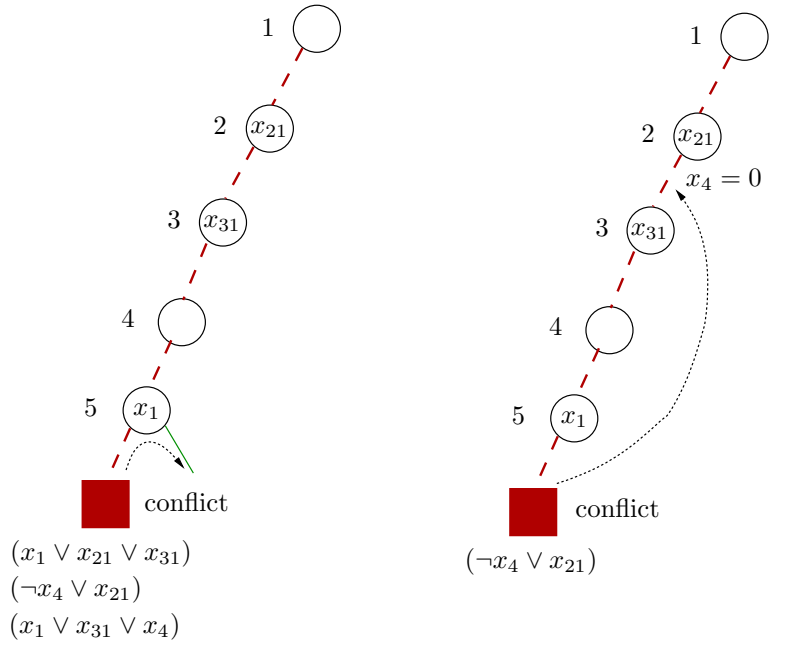
always stop clause learning at the *first UIP* [ZMMM01]. In addition, Chaff proposed to always backtrack given the information of the learnt clause. Observe that when each learnt clause is created it has all literals but one assigned value 0 [MSS96], and so it is unit (or *assertive*). Moreover, each learnt clause will remain unit until the search procedure backtracks to the highest decision level of its other literals. As a result, the authors of Chaff proposed to always take the backtrack step. This form of clause learning, and associated backtracking procedure, is referred to as *first UIP clause learning*. The modified clause learning scheme does not affect completeness [ZMMM01, Zha03], since each learnt clause is explained by a sequence of resolution steps, and is assertive when created. Existing results indicate that the first UIP clause learning procedure may end up doing more backtracking than the original clause learning of GRASP [ZMMM01]. However, Chaff creates significantly fewer clauses and is significantly more effective at backtracking. As a result, the first UIP learning scheme is now commonly used by the majority of CDCL SAT solvers.

Example 4.4.3 (Different Backtracking Schemes)**.** Figure 4.3 illustrates the two learning and backtracking schemes. In the GRASP [MSS96] learning and backtracking scheme, for the first conflict, the decision level of the conflict is kept (i.e. decision level 5). Backtracking only occurs if the resulting unit propagation yields another conflict. In contrast, in the Chaff [MMZ+01] learning and backtracking scheme, the backtrack step is *always* taken after each conflict. For the example in the figure, GRASP would proceed from decision level 5, whereas Chaff backtracks to decision level 2, and proceeds from decision level 2. One motivation for this difference is due to the data structures used. In GRASP backtracking is costly, and depends on the total number of literals, whereas in Chaff backtracking is inexpensive, and depends only on the assigned variables.

Figure 4.3 also illustrates the clauses learnt by GRASP and Chaff. Since Chaff stops at the first UIP, only a single clause is learnt. In contrast, GRASP learns clauses at all UIPs. Moreover, GRASP also learns a *global* clause, which it then uses either for forcing the second branch at the current decision level (in this case 5) or for backtracking. The global clause used in GRASP is optional and serves to ensure that at a given decision level both branches are made with respect to the same variable.

4.4.5. Uses of Clause Learning

Clause learning finds other applications besides the key efficiency improvements to CDCL SAT solvers. One example is *clause reuse* [MSS97, Str01]. In a large number of applications, clauses learnt for a given CNF formula can often be



(a) GRASP clause learning [MSS96] (b) Chaff clause learning [MMZ+01]

Figure 4.3. Alternative backtracking schemes

reused for related CNF formulas. Clause reuse is also used in incremental SAT, for example in algorithms for pseudo-Boolean optimization [ES06].

Moreover, for unsatisfiable subformulas, the clauses learnt by a CDCL SAT solver encode a resolution refutation of the original formula [ZM03, BKS04]. Given the way clauses are learnt in SAT solvers, each learnt clause can be explained by a number of resolution steps, each of which is a trivial resolution step [BKS04, Gel07]. As a result, the resolution refutation can be obtained from the learnt clauses in linear time and space on the number of learnt clauses [ZM03, Gel07].

For unsatisfiable formulas, the resolution refutations obtained from the clauses learnt by a SAT solver serve as a certificate for validating the correctness of the SAT solver. Moreover, resolution refutations based on clause learning find key practical applications, including hardware model checking [McM03].

Besides allowing producing a resolution refutation, learnt clauses also allow identifying a subset of clauses that is also unsatisfiable. For example, a *minimally unsatisfiable subformula* can be derived by iteratively removing a single clause and checking unsatisfiability [ZM03]. Unnecessary clauses are discarded, and eventually a minimally unsatisfiable subformula is obtained.

The success of clause learning in SAT motivated its use in a number of extensions of SAT. Clause learning has also been applied successfully in algorithms for binate covering [MMS00a, MMS02], pseudo-Boolean optimization [MMS00b, MMS04, ARMS02, CK03] , quantified Boolean formulas [Let02, ZM02, GNT02]

, and satisfiability modulo theories [GHN+04] . In binate covering and pseudo-Boolean optimization, clause learning has also been used for backtracking from bound conflicts associated with the cost function [MMS02, MMS04].

4.5. Modern CDCL Solvers

This section describes modern CDCL solvers. Apart from conflict analysis, these solvers include lazy data structures, search restarts, conflict-driven branching heuristics and clause deletion strategies.

4.5.1. Lazy Data Structures

Implementation issues for SAT solvers include the design of suitable data structures for storing clauses, variables and literals. The implemented data structures dictate the way BCP and conflict analysis are implemented and have a significant impact on the run time performance of the SAT solver. Recent state-of-the-art SAT solvers are characterized by using very efficient data structures, intended to reduce the CPU time required per each node in the search tree. Conversely, traditional SAT data structures are accurate, meaning that is possible to know exactly the value of each literal in the clause. Examples of traditional data structures, also called adjacency lists data structures, can be found in GRASP [MSS96], relsat [BS97] and satz [LA97]. Examples of the most recent SAT data structures, which are not accurate and therefore are called *lazy*, include the head/tail lists used in Sato [Zha97] and the watched literals used in Chaff [MMZ+01].

Traditional backtrack search SAT algorithms represent clauses as lists of literals, and associate with each variable x a list of the clauses that contain a literal in x. Clearly, after assigning a variable x the clauses with literals in x are immediately aware of the assignment of x. The lists associated with each variable can be viewed as containing the clauses that are *adjacent* to that variable. In general, we use the term *adjacency lists* to refer to data structures in which each variable x contains a *complete* list of the clauses that contain a literal in x.

These adjacency lists data structures share a common problem: each variable x keeps references to a potentially large number of clauses, which in CDCL SAT solvers often increase as the search proceeds. Clearly, this impacts negatively the amount of operations associated with assigning x. Moreover, it is often the case that most of x's clause references do not need to be analyzed when x is assigned, since most of the clauses do not become unit or unsatisfied. Observe that *lazily* declaring a clause to be satisfied does not affect the correctness of the algorithm.

Considering that only unsatisfied and unit clauses must be identified, it suffices to have two references for each clause. (Although additional references may be required to guarantee clauses' consistency after backtracking.) These references never reference literals assigned value 0. Hence, such references are allowed to move along the clause: whenever a referenced literal is assigned value 0, the reference moves to another literal either assigned value 1 or assigned value u (i.e. unassigned). Algorithm 2 shows how the value and the position of these two references (REFA and REFB) are enough for declaring a clause to be satisfied, unsatisfied or unit. As already mentioned, a clause is lazily declared to be satisfied,

Algorithm 2 Identifying clauses status with lazy data structures

```
CLAUSE_STATUS(ω)
1  if REFA(ω) == 1 or REFB(ω) == 1
2     then return SATISFIED
3     else if REFA(ω) == 0 and POSITION_REFA(ω) == POSITION_REFB(ω)
4             then return UNSATISFIED
5             else if REFA(ω) == u and POSITION_REFA(ω) == POSITION_REFB(ω)
6                     then return UNIT
7                     else return UNRESOLVED
```

meaning that some clauses being satisfied are not recognized as so (being identified as unresolved instead). Again, this aspect does not affect the correctness of the algorithm.

In this section we analyze *lazy* data structures, which are characterized by each variable keeping a reduced set of clauses' references, for each of which the variable can be effectively used for declaring the clause as unit, as satisfied or as unsatisfied.

4.5.1.1. The Head and Tail Data Structure

The first lazy data structure proposed for SAT was the *Head and Tail* (H/T) data structure, originally used in the Sato SAT solver [Zha97] and later described in [ZS00]. As the name implies, this data structure associates two references with each clause, the *head* (H) and the *tail* (T) literal references.

Initially the head reference points to the first literal, and the tail reference points to the last literal. Each time a literal pointed to by either the head or tail reference is assigned, a new unassigned literal is searched for. Both pointers move towards to the middle of the clause. In case an unassigned literal is identified, it becomes the new head (or tail) reference, and a *new* reference is created and associated with the literal's variable. These references guarantee that H/T positions are correctly recovered when the search backtracks. In case a satisfied literal is identified, the clause is declared satisfied. In case no unassigned literal can be identified, and the other reference is reached, then the clause is declared unit, unsatisfied or satisfied, depending on the value of the literal pointed to by the other reference.

When the search process backtracks, the references that have become associated with the head and tail references can be discarded, and the previous head and tail references become activated. Observe that this requires in the worst-case associating with each clause a number of literal references in variables that equals the number of literals.

This data structure is illustrated in figure 4.4(left). We illustrate the H/T data structure for one clause for a sequence of assignments. Each clause is represented by an array of literals. Literals have different representations depending on being unassigned, assigned value 0 (unsatisfied) or assigned value 1 (satisfied). Each assigned literal is associated with a decision level indicating the level where the literal was assigned. In addition, we represent the head (H) and tail

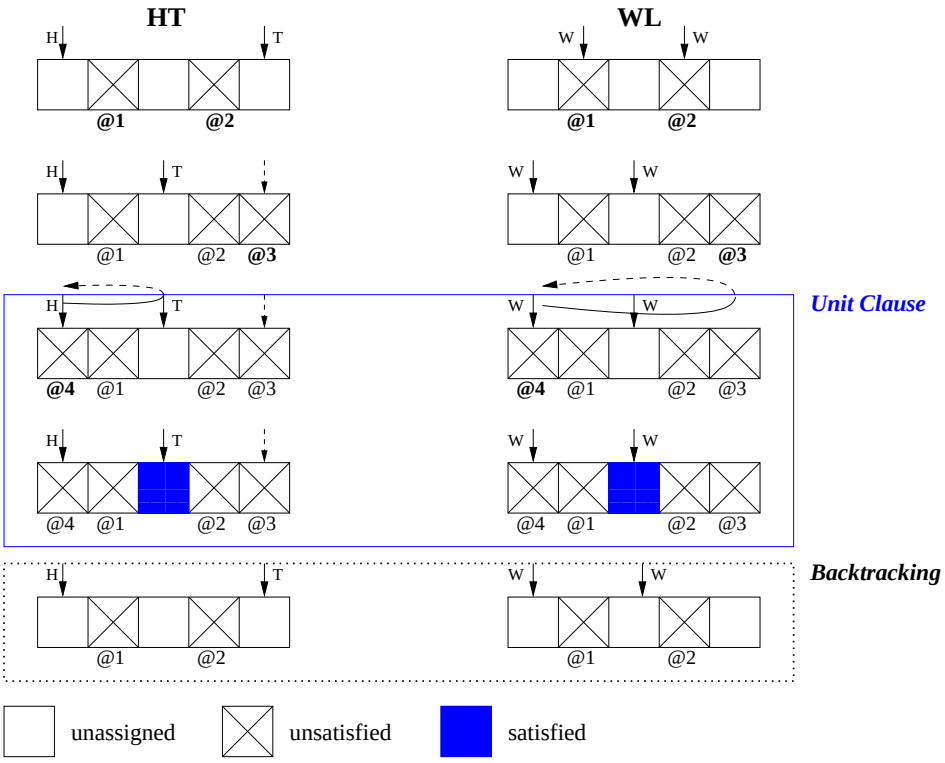


Figure 4.4. Operation of lazy data structures

(T) pointers that point to a specific literal. Initially, the H/T pointer points to the left/rightmost literal, respectively. These pointers only point to unassigned literals. Hence, each time one literal pointed by one of these pointers is assigned, the pointer has to move inwards. However, a new reference for the just assigned literal is created (represented with a dash line). When the two pointers reach the same unassigned literal the clause is unit. When the search backtracks, the H/T pointers must be moved. The pointers are now placed at its previous positions, i.e. at the position they were placed before being moved inwards.

4.5.1.2. The Watched Literal Data Structure

The more recent Chaff SAT solver [MMZ+01] proposed a new data structure, the Watched Literals (WL), that solves some of the problems posed by H/T lists. As with H/T lists, two references are associated with each clause. However, and in contrast with H/T lists, there is *no* order relation between the two references, allowing the references to move in any direction. The lack of *order* between the two references has the key advantage that no literal references need to be updated when backtracking takes place. In contrast, unit or unsatisfied clauses are identified only after traversing *all* the clauses' literals; a clear drawback. The identification of satisfied clauses is similar to H/T lists.

The most significant difference between H/T lists and watched literals occurs when the search process backtracks, in which case the references to the watched literals are not modified. Consequently, and in contrast with H/T lists, there is no need to keep additional references. This implies that for each clause the number of literal references that are associated with variables is kept *constant*.

This data structure is also illustrated in figure 4.4(right). The two watched literal pointers are undifferentiated as there is no order relation. Again, each time one literal pointed by one of these pointers is assigned, the pointer has to move inwards. However, in contrast with the H/T data structure, these pointers may move in both directions. This causes the whole clause to be traversed when the clause becomes unit. In addition, no references have to be kept to the just assigned literals, since pointers do not move when backtracking.

4.5.1.3. Variations of the Lazy Data Structures

Even though lazy data structures suffice for backtrack search SAT solvers that solely utilize BCP, the *laziness* of these data structures may pose some problems, in particular for algorithms that aim the integration of more advanced techniques for the identification of necessary assignments, namely restricted resolution, two-variable equivalence, and pattern-based clause inference, among other techniques [GW00, MS00, Bra01, Bac02]. For these techniques, it is essential to know which clauses become binary and/or ternary during the search. However, lazy data structures are not capable of keeping precise information about the set of binary and/or ternary clauses. Clearly, this can be done by associating additional literal references with each clause, which introduces additional overhead. Actually, the use of additional references has first been referred in [Gel02].

In addition we may use literal sifting [LMS05], along with two additional references, to dynamically rearrange the list of literals. Assigned variables are sorted by non-decreasing decision level, starting from the first or last literal reference, and terminating at the most recently assigned literal references. This sorting is achieved by sifting assigned literals as each is visited by one of the two usual references. The main disadvantage is the requirement to visit all literals between the two additional literal sifting references each time the clause is either unit or unsatisfied. (There could be only one literal sifting reference, even though the overhead of literal sifting then becomes more significant.) Observe that literal sifting may be implemented either in H/T data structures or in watched literals data structures.

Later on, a different data structure (Watched Lists with Conflict Counter, WLCC) was introduced with the purpose of combining the advantages of WL and sifting. Overall, WLCC has the advantage of reducing the number of literals to be visited by pointers [Nad02].

Another optimization is the special handling of the clauses that are more common in problem instances: binary and ternary clauses [LMS05, Rya04, PH02]. Both binary and ternary clauses can be identified as unit, satisfied or unsatisfied in constant time, thus eliminating the need for moving literal references around. Since the vast majority of the initial number of clauses for most real-world problem instances are either binary or ternary, the average CPU time required to handle each clause may be noticeably reduced. In this situation, lazy data structures are

Table 4.3. Comparison of the data structures

data structures			AL	HT	WL
lazy data structure?			N	Y	Y
# literal references	min		L	2C	2C
	max		L	L	2C
# visited literals	when identifying unit/unsat cl^s	min	1	1	W-1
		max	1	W-1	W-1
	when backtracking		L_b	L_b	0

L = number of literals
C = number of clauses
W = number of literals in clause
L_b = number of literals to be unassigned when backtracking

solely applied to original large clauses and to clauses recorded during the search process, which are known for having a huge number of literals.

4.5.1.4. A Comparison of Data Structures

Besides describing the organization of each data structure, it is also interesting to characterize each one in terms of the memory requirements and computational effort. Table 4.3 provides a comparison of the data structures described above, including the traditional data structures (Adjacency Literals, AL) and the lazy data structures (Head and Tail literals, HT, and Watched Literals, WL).

The table indicates which data structures are *lazy*, the minimum and maximum total number of literal references associated with all clauses, and also a broad indication of the work associated with keeping clause state when the search either moves forward (i.e. implies assignments) or backward (i.e. backtracks).

Even though it is straightforward to prove the results shown, a careful analysis of the behavior of each data structure is enough to establish these results. For example, when backtracking takes place, the WL data structure updates *no* literal references. Hence, the number of visited literal references for each conflict is 0.

4.5.2. Search Restarts

Rapid randomized restarts were introduced in complete search to eliminate heavy-tails [GSK98]. The heavy-tailed behaviour is characterized by a non-negligible probability of hitting a problem that requires exponentially more time to solve than any that has been encountered before [GSC97]. Randomization applied to variable and/or value selection ensures with high probability that different subtrees are searched each time the backtrack search algorithm is restarted.

Current state-of-the-art SAT solvers already incorporate random restarts [BMS00, MMZ$^+$01, GN02]. In these SAT solvers, variable selection heuristics are randomized and search restart strategies are used. Randomized restarts have first been shown to yield dramatic improvements on satisfiable random instances that exhibit heavy-tailed behavior [GSK98]. However, this strategy is also quite effective for real-world instances, including unsatisfiable instances [BMS00].

The use of restarts implies the use of additional techniques to guarantee completeness. Observe that restarting the search after a specific number of backtracks

(or alternatively a specific number of search nodes) may cause the search to become incomplete, as not enough search space may be provided for a solution to be found. A simple solution to this problem is to increase the cutoff to iteratively increase the search space [GSK98, BMS00]. In addition, in the context of CDCL SAT solvers the clauses learnt may be used. On the one hand, the clauses learnt in previous runs may also be useful in the next runs, as similar conflicts may arise. On the other hand, the number of learnt clauses to be kept may be used as the cutoff to be increased. If all clauses are kept then we may ensure that the search is complete. Even better, we may keep only one clause learnt on each iteration and this condition suffices to guarantee completeness [LMS07].

More recently, an empirical evaluation has been performed to understand the relation between the decision heuristic, backtracking scheme, and restart policy in the context of CDCL solvers [Hua07]. When the search restarts, the branching heuristic scores represent the solver's current state of belief about the order in which future decisions should be made. Hence, without the freedom of restarts, the solver would not be able to fully execute its belief because it is bound by the decisions that have been made earlier. This observation further motivates for the use of rapid randomized restarts. Indeed, the author has performed an empirical evaluation on using different restart schemes on a relevant set of industrial benchmarks and Luby's strategy [LSZ93] has shown the best performance.

4.5.3. Conflict-Driven Branching Heuristics

The early branching heuristics made use of all the information available from the data structures, namely the number of satisfied/unsatisfied and unassigned literals. These heuristics are updated during the search and also take into account the clauses that are learnt. An overview on such heuristics is provided in [MS99].

More recently, a different kind of variable selection heuristic (referred to as VSIDS, Variable State Independent Decaying Sum) has been proposed by Chaff authors [MMZ$^+$01]. One of the reasons for proposing this new heuristic was the introduction of *lazy* data structures, where the knowledge of the dynamic size of a clause is not accurate. Hence, the heuristics described above cannot be used.

VSIDS selects the literal that appears most frequently over all the clauses, which means that one counter is required for each one of the literals. Initially, all counters are set to zero. During the search, the metrics only have to be updated when a new recorded clause is created. More than to develop an *accurate* heuristic, the motivation has been to design a *fast* (but dynamically adapting) heuristic. In fact, one of the key properties of this strategy is the very low overhead, due to being independent of the variable state.

Two Chaff-like SAT solvers, BerkMin [GN02] and siege [Rya04], have improved the VSIDS heuristic. BerkMin also measures clauses' age and activity for deciding the next branching variable, whereas siege gives priority to assigning variables on recently recorded clauses.

4.5.4. Clause Deletion Strategies

Unrestricted clause recording can in some cases be impractical. Recorded clauses consume memory and repeated recording of clauses can eventually lead to the

exhaustion of the available memory. Observe that the number of recorded clauses grows with the number of conflicts; in the worst case, such growth can be *exponential* in the number of variables. Furthermore, large recorded clauses are known for not being particularly useful for search pruning purposes [MSS96]. Adding larger clauses leads to additional overhead for conducting the search process and, hence, it eventually costs more than what it saves in terms of backtracks.

As a result, there are three main solutions for guaranteeing the worst case growth of the recorded clauses to be *polynomial* in the number of variables:

1. We may consider *n-order learning*, that records only clauses with n or fewer literals [Dec90a].
2. Clauses can be temporarily recorded while they either imply variable assignments or are unit clauses, being discarded as soon as the number of unassigned literals is greater than an integer m. This technique is named *m-size relevance-based learning* [BS97].
3. Clauses with a size less than a threshold k are kept during the subsequent search, whereas larger clauses are discarded as soon as the number of unassigned literals is greater than one. We refer to this technique as *k-bounded learning* [MSS96].

Observe that k-bounded learning can be combined with m-size relevance-based learning. The search algorithm is organized so that all recorded clauses of size no greater than k are kept and larger clauses are deleted only after m literals have become unassigned.

More recently, a heuristic clause deletion policy has been introduced [GN02]. Basically, the decision whether a clause should be deleted is based not only on the number of literals but also on its *activity* in contributing to conflict making and on the number of decisions taken since its creation.

4.5.5. Additional Topics

Other relevant techniques include the use of preprocessing aiming at reducing the formula size in order to speedup overall SAT solving time [EB05]. These techniques are expected to eliminate variables and clauses using an efficient implementation.

An additional topic is the use of early conflict detection BCP with an implication queue sorting [LSB05]. The idea is to use a heuristic to select which implications to check first from the implication queue. This is in contrast to other solvers which process the implication queue in chronological order.

More recently, it has been shown that modern CDCL solvers implicitly build and prune a decision tree whose nodes are associated with flipped variables. This motivates a practical useful enhancement named local conflict-clause recording [DHN07]. A local conflict clause is a non-asserting conflict clause, recorded in addition to the first UIP conflict clause if the last decision level contains some flipped variables that have implied the conflict. The last variable in these circumstances is considered to be a decision variable, defining a new decision level. A local conflict clause is the first UIP clause with respect to this new decision level.

A potential disadvantage of CDCL SAT solvers is that all decision and implied variable assignments made between the level of the conflict and the backtracking

level are erased during backtracking. One possible solution is to simply save the partial decision assignments [PD07]. When the solver decides to branch on a variable, it first checks whether there is information saved with respect to this variable. In case it exists, the solver assigns the variable with the same value that has been used in the past. In case it does not exist, the solver uses the default phase selection heuristic.

4.6. Bibliographical and Historical Notes

Learning from conflicting conditions has a long history. Dependency-directed backtracking was first proposed by Stallman and Sussman [SS77], and was used and extended in a number of areas, including Truth Maintenance Systems [Doy79] and Logic Programming [PP80, Bru81]. In Constraint Programming, original work addressed conditions for non-chronological backtracking [Gas77]. More recent work addressed learning [Dec90b, Gin93, FD94] and alternative forms of non-chronological backtracking [Dec90b, Pro93].

The use of learning and associated non-chronological backtracking in SAT was first proposed in the mid 90s [MS95, MSS96]. Later independent work also proposed clause learning for SAT [BS97].

The original work on using clause learning in SAT was inspired by earlier work in other areas. However, key new ideas were proposed in the GRASP SAT solver [MSS96, MSS99], that explain the success of modern SAT solvers. Besides implementing clause learning and non-chronological backtracking, one key aspect of GRASP was its ability for exploiting the *structure* of implied assignments provided by unit propagation. This idea allows learning much smaller clauses, many of which end up being unrelated with most branching steps made during DPLL. This ability for exploiting the intrinsic structure of practical SAT problems is the main reason for the success of CDCL SAT solvers.

Besides clause learning based on unit propagation, GRASP also proposed UIPs, another hallmark of modern SAT solvers. UIPs represent dominators in the implication graph, and can be viewed as another technique for further exploiting the structure of unit propagation. UIPs were inspired by the other form of dominators in circuit testing, the Unique Sensitization Points (USPs) [MSS94, MS95].

Despite the original success of CDCL SAT solvers, the application of SAT to a number of strategic applications in the late 90s, including model checking and model finding, motivated the development of more effective SAT solvers. The outcome was Chaff [MMZ$^+$01]. Chaff also proposed a number of contributions that are now key to all modern CDCL SAT solvers, including the *watched literals* lazy data structure, the conflict-inspired VSIDS branching heuristic, and the first-UIP backtracking scheme [ZMMM01].

Finally, there are several recent contributions that have been used the best performing SAT solvers in the SAT competitions [LSR], namely simplification of CNF formulas [EB05], priority queue for unit propagation [LSB05], lightweight component caching [PD07], and more complex clause learning schemes [DHN07].

References

[ARMS02] F. Aloul, A. Ramani, I. Markov, and K. A. Sakallah. Generic ILP versus specialized 0-1 ILP: An update. In *International Conference on Computer-Aided Design*, pages 450–457, November 2002.

[Bac02] F. Bacchus. Exploiting the computational tradeoff of more reasoning and less searching. In *Fifth International Symposium on Theory and Applications of Satisfiability Testing*, pages 7–16, May 2002.

[BKS04] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[BMS00] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 489–494, September 2000.

[Bra01] R. I. Brafman. A simplifier for propositional formulas with many binary clauses. In *International Joint Conference on Artificial Intelligence*, August 2001.

[Bru81] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[BS97] R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *National Conference on Artificial Intelligence*, pages 203–208, July 1997.

[CK03] D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. In *Design Automation Conference*, pages 830–835, June 2003.

[Dec90a] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.

[Dec90b] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[DHN07] N. Dershowitz, Z. Hanna, and A. Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In *International Conference in Theory and Applications of Satisfiability Testing*, pages 287–293, 2007.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, July 1960.

[EB05] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *International Conference in Theory and Applications of Satisfiability Testing*, pages 61–75, 2005.

[ES06] N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computa-*

tion, 2:1–26, March 2006.

[FD94] D. Frost and R. Dechter. Dead-end driven learning. In *National Conference on Artificial Intelligence*, pages 294–300, 1994.

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

[Gel02] A. Van Gelder. Generalizations of watched literals for backtracking search. In *International Symposium on Artificial Intelligence and Mathematics*, January 2002.

[Gel07] A. Van Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 328–333, 2007.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Computer-Aided Verification*, pages 175–188, 2004.

[Gin93] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[GN02] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Design, Automation and Testing in Europe Conference*, pages 142–149, March 2002.

[GNT02] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified Boolean logic satisfiability. In *National Conference on Artificial Intelligence*, pages 649–654, August 2002.

[GSC97] C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *International Conference on Principles and Practice of Constraint Programming*, pages 121–135, 1997.

[GSK98] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *National Conference on Artificial Intelligence*, pages 431–437, July 1998.

[GW00] J. F. Groote and J. P. Warners. The propositional formula checker heerhugo. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT 2000*, pages 261–281. IOS Press, 2000.

[Hua07] J. Huang. The effect of restarts on clause learning. In *International Joint Conference on Artificial Intelligence*, pages 2318–2323, 2007.

[LA97] C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 341–355, October 1997.

[Let02] R. Letz. Lemma and model caching in decision procedures for quantified Boolean formulas. In *International Conference on Automated Reasoning with Analytic Tableau x and Related Methods*, pages 160–175, July 2002.

[LMS05] I. Lynce and J. P. Marques-Silva. Efficient data structures for backtrack search SAT solvers. *Annals of Mathematics and Artificial Intelligence*, 47(1):137–152, January 2005.

[LMS07] I. Lynce and J. P. Marques-Silva. Random backtracking in backtrack search algorithms for satisfiability. *Discrete Applied Mathematics*, 155(12):1604–1612, 2007.

[LSB05] M. D. T. Lewis, T. Schubert, and B. Becker. Speedup techniques utilized in modern SAT solvers. In *International Conference in Theory and Applications of Satisfiability Testing*, pages 437–443, 2005.

[LSR] D. Le Berre, L. Simon, and O. Roussel. SAT competition. www.satcompetition.org.

[LSZ93] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[McM03] K. L. McMillan. Interpolation and SAT-based model checking. In *Computer-Aided Verification*, 2003.

[MMS00a] V. Manquinho and J. P. Marques-Silva. On using satisfiability-based pruning techniques in covering algorithms. In *Design, Automation and Test in Europe Conference*, pages 356–363, March 2000.

[MMS00b] V. Manquinho and J. P. Marques-Silva. Search pruning conditions for Boolean optimization. In *European Conference on Artificial Intelligence*, pages 130–107, August 2000.

[MMS02] V. M. Manquinho and J. P. Marques-Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on Computer-Aided Design*, 21(5):505–516, May 2002.

[MMS04] V. Manquinho and J. P. Marques-Silva. Satisfiability-based algorithms for Boolean optimization. *Annals of Mathematics and Artificial Intelligence*, 40(3-4), March 2004.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, June 2001.

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

[MS99] J. P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the Portuguese Conference on Artificial Intelligence*, pages 62–74, September 1999.

[MS00] J. P. Marques-Silva. Algebraic simplification techniques for propositional satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 537–542, September 2000.

[MSS94] J. P. Marques-Silva and K. A. Sakallah. Dynamic search-space pruning techniques in path sensitization. In *Design Automation Conference*, pages 705–711, June 1994.

[MSS96] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[MSS97] J. P. Marques-Silva and K. A. Sakallah. Robust search algorithms for test pattern generation. In *Fault-Tolerant Computinig Symposium*, pages 152–161, June 1997.

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

[MSS00] J. P. Marques-Silva and K. A. Sakallah. Invited tutorial: Boolean satisfiability algorithms and applications in electronic design automation. In *Computer Aided Verification*, page 3, 2000.

[Nad02] A. Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew University of Jerusalem, November 2002.

[PD07] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *International Conference in Theory and Applications of Satisfiability Testing*, pages 294–299, 2007.

[PH02] S. Pilarski and G. Hu. SAT with partial clauses and back-leaps. In *Design Automation Conference*, pages 743–746, 2002.

[PP80] L. Moniz Pereira and A. Porto. Selective bachtracking for logic programs. In *Conference on Automated Deduction*, pages 306–317, 1980.

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

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

[Str01] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 58–70, 2001.

[Tar74] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 1974.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, July 1997.

[Zha03] L. Zhang. *Searching the Truth: Techniques for Satisfiability of Boolean Formulas*. PhD thesis, Princeton University, June 2003.

[ZM88] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *National Conference on Artificial Intelligence*, pages 155–160, July 1988.

[ZM02] L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *International Conference on Computer-Aided Design*, pages 442–449, 2002.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Testing in Europe Conference*, pages 10880–10885, 2003.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *International Conference on Computer-Aided Design*, pages 279–285, 2001.

[ZS00] H. Zhang and M. Stickel. Implementing the Davis-Putnam method. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT 2000*, pages 309–326. IOS Press, 2000.