2

Decision Procedures for Propositional Logic

2.1 Propositional Logic

We assume that the reader is familiar with propositional logic, and with the complexity classes NP and NP-complete.

The syntax of formulas in propositional logic is defined by the following grammar:

$$\begin{aligned} \textit{formula} &: \textit{formula} \wedge \textit{formula} \mid \neg \textit{formula} \mid (\textit{formula}) \mid \textit{atom} \\ \textit{atom} &: \textit{Boolean-identifier} \mid \text{TRUE} \mid \text{FALSE} \end{aligned}$$

Other Boolean operators such as OR ($\vee$) and XOR ($\oplus$) can be constructed using AND ($\wedge$) and NOT ($\neg$).

2.1.1 Motivation

Since SAT, the problem of deciding the satisfiability of propositional formulas, is NP-complete, it can be used for solving any NP problem. Any other NP-complete problem (e.g., k-coloring of a graph) can be used just as well, but none of them has a natural input language such as propositional logic to model the original problem. Indeed, propositional logic is widely used in diverse areas such as database queries, planning problems in artificial intelligence, automated reasoning, and circuit design. Let us consider two examples: a layout problem and a program verification problem.

Example 2.1. Let $S = \{s_1, \ldots, s_n\}$ be a set of radio stations, each of which has to be allocated one of k transmission frequencies, for some $k < n$. Two stations that are too close to each other cannot have the same frequency. The set of pairs having this constraint is denoted by E. To model this problem, define a set of propositional variables $\{x_{ij} \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, k\}\}$. Intuitively, variable x_{ij} is set to TRUE if and only if station i is assigned the frequency j. The constraints are:

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_2

- Every station is assigned at least one frequency:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} x_{ij} \,. \tag{2.1}$$

- Every station is assigned not more than one frequency:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k-1} (x_{ij} \implies \bigwedge_{j<t\leq k} \neg x_{it}) \,. \tag{2.2}$$

- Close stations are not assigned the same frequency. For each $(i, j) \in E$,

$$\bigwedge_{t=1}^{k} (x_{it} \implies \neg x_{jt}) \,. \tag{2.3}$$

Note that the input of this problem can be represented by a graph, where the stations are the graph's nodes and E corresponds to the graph's edges. Checking whether the allocation problem is solvable corresponds to solving what is known in graph theory as the *k-colorability* problem: can all nodes be assigned one of k colors such that two adjacent nodes are assigned different colors? Indeed, one way to solve k-colorability is by reducing it to propositional logic. ▟

Example 2.2. Consider the two code fragments in Fig. 2.1. The fragment on the right-hand side might have been generated from the fragment on the left-hand side by an optimizing compiler.

```
if(!a && !b) h();
else
    if(!a) g();
    else f();
```

```
if(a) f();
else
    if(b) g();
    else h();
```

Fig. 2.1. Two code fragments—are they equivalent?

We would like to check if the two programs are equivalent. The first step in building the **verification condition** is to model the variables a and b and the procedures that are called using the Boolean variables a, b, f, g, and h, as can be seen in Fig. 2.2.

The `if-then-else` construct can be replaced by an equivalent propositional logic expression as follows:

$$(\texttt{if } x \texttt{ then } y \texttt{ else } z) \quad \equiv \quad (x \wedge y) \vee (\neg x \wedge z) \,. \tag{2.4}$$

Consequently, the problem of checking the equivalence of the two code fragments is reduced to checking the validity of the following propositional formula:

```
if ¬a ∧ ¬b then h
else
   if ¬a then g
   else f
```

```
if a then f
else
   if b then g
   else h
```

Fig. 2.2. In the process of building a formula—the verification condition—we replace the program variables and the function symbols with new Boolean variables

$$\begin{array}{ll} & (\neg a \wedge \neg b) \wedge h \ \vee \ \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \ \vee \ a \wedge f) \\ \iff & a \wedge f \ \vee \ \neg a \wedge (b \wedge g \ \vee \ \neg b \wedge h) \ . \end{array} \tag{2.5}$$

2.2 SAT Solvers

2.2.1 The Progress of SAT Solving

Given a propositional formula $\mathcal{B}$, a SAT solver decides whether $\mathcal{B}$ is satisfiable; if it is, it also reports a satisfying assignment. In this chapter, we consider only the problem of solving formulas in conjunctive normal form (CNF) (see Definition 1.20). Since every formula can be converted to this form in linear time (as explained right after Definition 1.20), this does not impose a real restriction.[1] Solving general propositional formulas can be somewhat more efficient in some problem domains, but most of the solvers and most of the research are still focused on CNF formulas.

The practical and theoretical importance of the satisfiability problem has led to a vast amount of research in this area, which has resulted in exceptionally powerful SAT solvers. Modern SAT solvers can solve many real-life CNF formulas with hundreds of thousands or even millions of variables in a reasonable amount of time. Figures 2.3 and 2.4 illustrate the progress of these tools through the years (see captions). Of course, there are also instances of problems two orders of magnitude smaller that these tools still cannot solve. In general, it is very hard to predict which instance is going to be hard to solve, without actually attempting to solve it. Some tools, however, called **SAT portfolio** solvers, use machine-learning techniques to extract features of CNF formulas in order to select the most suitable SAT solver for the job. More details on this approach are given in Sect. 2.4.

The success of SAT solvers can be largely attributed to their ability to learn from wrong assignments, to prune large search spaces quickly, and to focus first on the "important" variables, those variables that, once given the

[1] Appendix B provides a library for performing this conversion and generating CNF in the DIMACS format, which is used by virtually all publicly available SAT solvers.

right value, simplify the problem immensely.[2] All of these factors contribute to the fast solving of both satisfiable and unsatisfiable instances. There is empirical evidence in [213] that shows that solving satisfiable instances fast requires a different set of heuristics than those that are necessary for solving unsatisfiable instances.

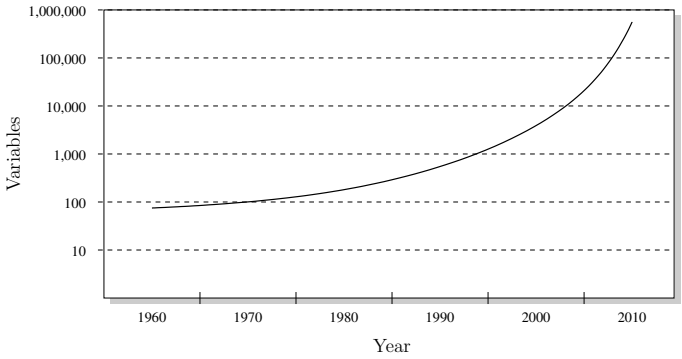

Fig. 2.3. The size of industrial CNF formulas (instances generated for solving various realistic problems such as verification of circuits and planning problems) that are regularly solved by SAT solvers in a few hours, according to year. Most of the progress in efficiency has been made in the last decade

The majority of modern SAT solvers can be classified into two main categories. The first category is based on the Conflict-Driven Clause Learning (**CDCL**) framework: in this framework the tool can be thought of as traversing and backtracking on a binary tree, in which internal nodes represent partial assignments, and the leaves represent full assignments. Building a simple CDCL solver is surprisingly easy: one can do so with fewer than 500 lines of C++ and STL.

The second category is based on a **stochastic search**: the solver guesses a full assignment, and then, if the formula is evaluated to FALSE under this assignment, starts to flip values of variables according to some (greedy) heuristic. Typically it counts the number of unsatisfied clauses and chooses the flip that minimizes this number. There are various strategies that help such solvers avoid local minima and avoid repeating previous bad moves. CDCL solvers, however, are considered better in most cases according to annual competitions that measure their performance with numerous CNF instances. CDCL solvers also have the advantage that, unlike most stochastic search methods, they are complete (see Definition 1.6). Stochastic methods seem to have an average

[2] Specifically, every formula has what is known as **backdoor variables** [284], which are variables that, once given the right value, simplify the formula to the point that it is polynomial to solve.

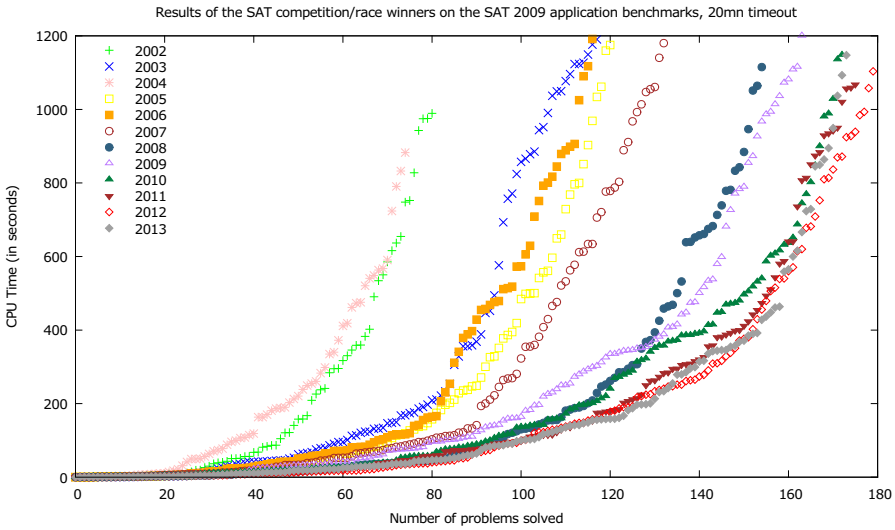

Fig. 2.4. Annual competitions measure the success of SAT solvers when applied to randomly selected benchmarks arriving from industry. The graph shows a comparison between the winners of these competitions as of 2002, when applied to a common benchmark set and using the same single-core hardware. Such graphs are nicknamed "cactus plots". A point (x, y) means that x benchmarks are solved within y amount of time each. Hence, the more the graph is to the right, the better it is. One may observe that the number of solved instances within 20 minutes has more than doubled within a decade, thanks to better algorithms. The instances in this set are large, and solvers created before 2002 run out of memory when trying to solve them. (Courtesy of Daniel Le-Berre)

advantage in solving randomly generated (satisfiable) CNF instances, which is not surprising: in these instances there is no structure to exploit and learn from, and no obvious choices of variables and values, which makes the heuristics adopted by CDCL solvers ineffective. We shall focus on CDCL solvers only.

A historical note: CDCL was developed over time as a series of improvements to the *Davis–Putnam–Loveland–Logemann* (**DPLL**) framework. See the bibliographic notes at the end of this chapter for further discussion.

2.2.2 The CDCL Framework

In its simplest form, a CDCL solver progresses by making a decision about a variable and its value, propagating implications of this decision that are easy to detect, and backtracking in the case of a conflict. Viewing the process as a search on a binary tree, each decision is associated with a **decision level**, which is the depth in the binary decision tree at which it is made, starting

from 1. The assignments implied by a decision are associated with its decision level. Assignments implied regardless of the current assignments (owing to **unary clauses**, which are clauses with a single literal) are associated with decision level 0, also called the **ground level**.

Definition 2.3 (state of a clause under an assignment). *A clause is* **satisfied** *if one or more of its literals are satisfied (see Definition 1.12),* **conflicting** *if all of its literals are assigned but not satisfied,* **unit** *if it is not satisfied and all but one of its literals are assigned, and* **unresolved** *otherwise.*

Note that the definitions of a unit clause and an unresolved clause are only relevant for partial assignments (see Definition 1.1).

Example 2.4. Given the partial assignment

$$\{x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_4 \mapsto 1\}\,, \tag{2.6}$$

$(x_1 \vee x_3 \vee \neg x_4)$	is satisfied,
$(\neg x_1 \vee x_2)$	is conflicting,
$(\neg x_1 \vee \neg x_4 \vee x_3)$	is unit,
$(\neg x_1 \vee x_3 \vee x_5)$	is unresolved.

Given a partial assignment under which a clause becomes unit, it must be extended so that it satisfies the unassigned literal of this clause. This observation is known as the **unit clause rule**. Following this requirement is necessary but obviously not sufficient for satisfying the formula.

For a given unit clause C with an unassigned literal l, we say that l is implied by C and that C is the **antecedent clause** of l, denoted by $\mathit{Antecedent}(l)$. If more than one unit clause implies l, the clause that the SAT solver actually used in order to imply l is the one we refer to as l's antecedent.

Example 2.5. The clause $C := (\neg x_1 \vee \neg x_4 \vee x_3)$ and the partial assignment $\{x_1 \mapsto 1,\ x_4 \mapsto 1\}$ imply the assignment x_3 and $\mathit{Antecedent}(x_3) = C$.

A framework followed by most modern CDCL solvers has been presented by, for example, Zhang and Malik [299], and is shown in Algorithm 2.2.1. The table in Fig. 2.6 includes a description of the main components used in this algorithm, and Fig. 2.5 depicts the interaction between them. A description of the Analyze-Conflict function is delayed to Sect. 2.2.6.

2.2.3 BCP and the Implication Graph

We now demonstrate Boolean constraint propagation (BCP), reaching a conflict, and backtracking. Each assignment is associated with the decision level

Algorithm 2.2.1: CDCL-SAT

Input: A propositional CNF formula $\mathcal{B}$
Output: "Satisfiable" if the formula is satisfiable and "Unsatisfiable" otherwise

```
1. function CDCL
2.     while (TRUE) do
3.         while (BCP() = "conflict") do
4.             backtrack-level := ANALYZE-CONFLICT();
5.             if backtrack-level < 0 then return "Unsatisfiable";
6.             BackTrack(backtrack-level);
7.         if ¬DECIDE() then return "Satisfiable";
```

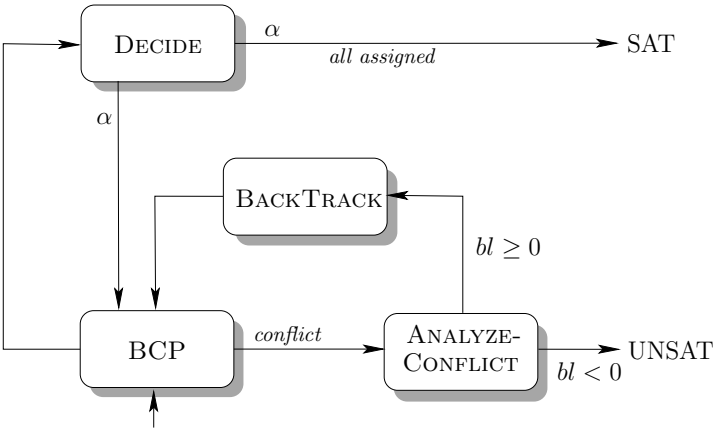

Fig. 2.5. CDCL-SAT: high-level overview of the Conflict-Driven Clause-Learning algorithm. The variable *bl* is the backtracking level, i.e., the decision level to which the procedure backtracks. α is an assignment (either partial or full)

at which it occurred. If a variable x_i is assigned 1 (TRUE) (owing to either a decision or an implication) at decision level dl, we write $x_i@dl$. Similarly, $\neg x_i@dl$ reflects an assignment of 0 (FALSE) to this variable at decision level dl. Where appropriate, we refer only to the truth assignment, omitting the decision level, in order to make the notation simpler.

$x_i@dl$

The process of BCP is best illustrated with an **implication graph**. An implication graph represents the current partial assignment and the reason for each of the implications.

Definition 2.6 (implication graph). *An* implication graph *is a labeled directed acyclic graph $G(V, E)$, where:*

Name	DECIDE()
Output	FALSE if and only if there are no more variables to assign.
Description	Chooses an unassigned variable and a truth value for it.
Comments	There are numerous heuristics for making these decisions, some of which are described later in Sect. 2.2.5. Each such decision is associated with a decision level, which can be thought of as the depth in the search tree.
Name	BCP()
Output	"conflict" if and only if a conflict is encountered.
Description	Repeated application of the unit clause rule until either a conflict is encountered or there are no more implications.
Comments	This repeated process is called Boolean Constraint Propagation (BCP). BCP is applied even before the first decision because of the possible existence of unary clauses.
Name	ANALYZE-CONFLICT()
Output	Minus 1 if a conflict at decision level 0 is detected (which implies that the formula is unsatisfiable). Otherwise, a decision level which the solver should backtrack to.
Description	A detailed description of this function is delayed to Sect. 2.2.4. Briefly, it is responsible for computing the backtracking level, detecting global unsatisfiability, and adding new constraints on the search in the form of new clauses.
Name	BACKTRACK(dl)
Description	Sets the current decision level to dl and erases assignments at decision levels larger than dl.

Fig. 2.6. A description of the main components of Algorithm 2.2.1

- *V represents the literals of the current partial assignment (we refer to a node and the literal that it represents interchangeably). Each node is labeled with the literal that it represents and the decision level at which it entered the partial assignment.*
- *E with $E = \{(v_i, v_j) \mid v_i, v_j \in V, \neg v_i \in Antecedent(v_j)\}$ denotes the set of directed edges where each edge (v_i, v_j) is labeled with $Antecedent(v_j)$.*
- *G can also contain a single* **conflict node** *labeled with κ and incoming edges $\{(v, \kappa) \mid \neg v \in c\}$ labeled with c for some conflicting clause c.*

The root nodes of an implication graph correspond to decisions, and the internal nodes to implications through BCP. A conflict node with incoming edges labeled with c represents the fact that the BCP process has reached a conflict, by assigning 0 to all the literals in the clause c (i.e., c is conflicting).

In such a case, we say that the graph is a **conflict graph**. The implication graph corresponds to all the decision levels lower than or equal to the current one, and is dynamic: backtracking removes nodes and their incoming edges, whereas new decisions, implications, and conflict clauses increase the size of the graph.

The implication graph is sensitive to the order in which the implications are propagated in BCP, which means that the graph is not unique for a given partial assignment. In most SAT solvers, this order is rather arbitrary (in particular, BCP progresses along a list of clauses that contain a given literal, and the order of clauses in this list can be sensitive to the order of clauses in the input CNF formula). In some other SAT solvers—see for example [223]—this order is not arbitrary; rather, it is biased towards reaching a conflict faster.

A **partial implication graph** is a subgraph of an implication graph, which illustrates the BCP at a specific decision level. Partial implication graphs are sufficient for describing Analyze-Conflict. The roots in such a partial graph represent assignments (not necessarily decisions) at decision levels lower than dl, in addition to the decision at level dl, and internal nodes correspond to implications at level dl. The description that follows uses mainly this restricted version of the graph.

Consider, for example, a formula that contains the following set of clauses, among others:

$$\begin{aligned} c_1 &= (\neg x_1 \vee x_2)\ , \\ c_2 &= (\neg x_1 \vee x_3 \vee x_5)\ , \\ c_3 &= (\neg x_2 \vee x_4)\ , \\ c_4 &= (\neg x_3 \vee \neg x_4)\ , \\ c_5 &= (x_1 \vee x_5 \vee \neg x_2)\ , \\ c_6 &= (x_2 \vee x_3)\ , \\ c_7 &= (x_2 \vee \neg x_3)\ , \\ c_8 &= (x_6 \vee \neg x_5)\ . \end{aligned} \tag{2.7}$$

Assume that at decision level 3 the decision was $\neg x_6@3$, which implied $\neg x_5@3$ owing to clause c_8 (hence, $Antecedent(\neg x_5) = c_8$). Assume further that the solver is now at decision level 6 and assigns $x_1 \mapsto 1$. At decision levels 4 and 5, variables other than $x_1, \ldots, x_6$ were assigned, and are not listed here as they are not relevant to these clauses.

The implication graph on the left of Fig. 2.7 demonstrates the BCP process at the current decision level 6 until, in this case, a conflict is detected. The roots of this graph, namely $\neg x_5@3$ and $x_1@6$, constitute a sufficient condition for creating this conflict. Therefore, we can safely add to our formula the **conflict clause**

$$c_9 = (x_5 \vee \neg x_1)\ . \tag{2.8}$$

While c_9 is logically implied by the original formula and therefore does not change the result, it prunes the search space. The process of adding conflict clauses is generally referred to as **learning**, reflecting the fact that this is the

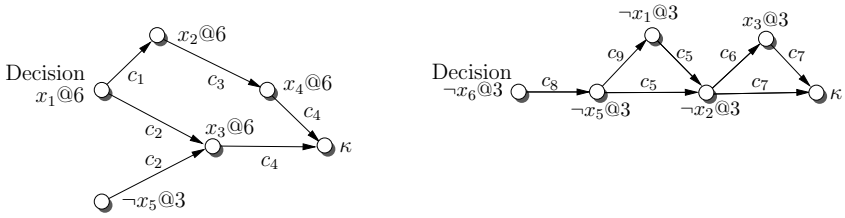

Fig. 2.7. A partial implication graph for decision level 6, corresponding to the clauses in (2.7), after a decision $x_1 \mapsto 1$ (*left*) and a similar graph after learning the conflict clause $c_9 = (x_5 \vee \neg x_1)$ and backtracking to decision level 3 (*right*)

solver's way to learn from its past mistakes. As we progress in this chapter, it will become clear that conflict clauses not only prune the search space, but also have an impact on the decision heuristic, the backtracking level, and the set of variables implied by each decision.

Analyze-Conflict is the function responsible for deriving new conflict clauses and computing the backtracking level. It traverses the implication graph backwards, starting from the conflict node κ, and generates a conflict clause through a series of steps that we describe later in Sect. 2.2.4. For now, assume that c_9 is indeed the clause generated.

After detecting the conflict and adding c_9, the solver determines which decision level to backtrack to according to the **conflict-driven backtracking** strategy. According to this strategy, the backtracking level is set to the *second most recent decision level in the conflict clause*, while erasing all decisions and implications made *after* that level. There are two special cases: when learning a unary clause, the solver backtracks to the ground level; when the conflict is *at* the ground level, the backtracking level is set to –1 and the solver exits and declares the formula to be unsatisfiable.

In the case of c_9, the solver backtracks to decision level 3 (the decision level of x_5), and erases all assignments from decision level 4 onwards, including the assignments to x_1, x_2, x_3, and x_4.

The newly added conflict clause c_9 becomes a unit clause since $x_5 = 0$, and therefore the assignment $\neg x_1@3$ is implied. This new implication restarts the BCP process at level 3. Clause c_9 is a special kind of conflict clause, called an **asserting clause**: it forces an immediate implication after backtracking. Analyze-Conflict can be designed to generate asserting clauses only, as indeed most competitive solvers do.

After asserting $x_1 = 0$ the solver again reaches a conflict, as can be seen in the right drawing in Fig. 2.7. This time the conflict clause (x_2) is added, and the solver backtracks to decision level 0 and continues from there. Why (x_2)? The strategy of Analyze-Conflict in generating these clauses is explained later in Sect. 2.2.4, but observe for the moment how indeed $\neg x_2$ leads to a conflict through clauses c_6 and c_7, as can also be inferred from Fig. 2.7 (right).

Aside: Multiple Conflict Clauses
More than one conflict clause can be derived from a conflict graph. In the present example, the assignment $\{x_2 \mapsto 1,\ x_3 \mapsto 1\}$ is also a sufficient condition for the conflict, and hence $(\neg x_2 \vee \neg x_3)$ is also a conflict clause. A generalization of this observation requires the following definition.

Definition 2.7 (separating cut). *A* separating cut *in a conflict graph is a minimal set of edges whose removal breaks all paths from the root nodes to the conflict node.*

This definition is applicable to a full implication graph (see Definition 2.6), as well as to a partial graph focused on the decision level of the conflict. The cut bipartitions the nodes into the *reason* side (the side that includes all the roots) and the *conflict* side. The set of nodes on the reason side that have at least one edge to a node on the conflict side constitute a sufficient condition for the conflict, and hence their negation is a legitimate conflict clause. Different SAT solvers have different strategies for choosing the conflict clauses that they add: some add as many as possible (corresponding to many different cuts), while others try to find the most effective ones. Some, including most of the modern SAT solvers, add a single clause, which is an asserting clause (see below), for each conflict. Modern solvers also have a strategy for *erasing* conflict clauses: without this feature the memory is quickly filled with millions of clauses. A typical strategy is to measure the *activity* of each clause, and periodically erase clauses with a low activity score. The activity score of a clause is increased when it participates in inferring new clauses.

Conflict-driven backtracking raises several issues:

- *It seems to waste work*, because the partial assignments up to decision level 5 can still be part of a satisfying assignment. However, empirical evidence shows that conflict-driven backtracking, coupled with a conflict-driven decision heuristic such as VSIDS (discussed later in Sect. 2.2.5), performs very well. A possible explanation for the success of this heuristic is that the conflict encountered can influence the decision heuristic to decide values or variables different from those at deeper decision levels (levels 4 and 5 in this case). Thus, keeping the decisions and implications made before the new information (i.e., the new conflict clause) had arrived may skew the search to areas not considered best anymore by the heuristic. Some of the wasted work can be saved, however, by simply keeping the last assignment, and reusing it whenever the variable's value has to be decided again [261]. An extensive analysis of this technique can be found in [222].
- *Is this process guaranteed to terminate?* In other words, how do we know that a partial assignment cannot be repeated forever? The learned conflict clauses cannot be the reason, because in fact most SAT solvers erase many

of them after a while to prevent the formula from growing too much. The reason is the following:

Theorem 2.8. *It is never the case that the solver enters decision level dl again with the same partial assignment.*

Proof. Consider a partial assignment up to decision level $dl - 1$ that does not end with a conflict, and assume falsely that this state is repeated later, after the solver backtracks to some lower decision level dl^- ($0 \leq dl^- < dl$). Any backtracking from a decision level dl^+ ($dl^+ \geq dl$) to decision level dl^- adds an implication at level dl^- of a variable that was assigned at decision level dl^+. Since this variable has not so far been part of the partial assignment up to decision level dl, once the solver reaches dl again, it is with a different partial assignment, which contradicts our assumption. ∎

The (hypothetical) progress of a SAT solver based on this strategy is illustrated in Fig. 2.8. More details of this graph are explained in the caption.

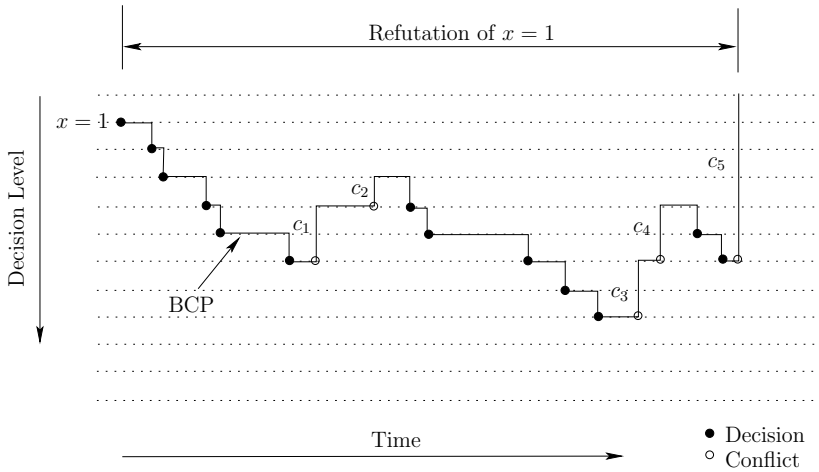

Fig. 2.8. Illustration of the progress of a SAT solver based on conflict-driven backtracking. Every conflict results in a conflict clause (denoted by $c_1, \ldots, c_5$ in the drawing). If the top left decision is $x = 1$, then this drawing illustrates the work done by the SAT solver to refute this wrong decision. Only some of the work during this time was necessary for creating c_5, refuting this decision, and computing the backtracking level. The "wasted work" (which might, after all, become useful later on) is due to the imperfection of the decision heuristic

2.2.4 Conflict Clauses and Resolution

Now consider Analyze-Conflict (Algorithm 2.2.2). The description of the algorithm so far has relied on the fact that the conflict clause generated is

an asserting clause, and we therefore continue with this assumption when considering the termination criterion for line 3. The following definitions are necessary for describing this criterion:

Algorithm 2.2.2: ANALYZE-CONFLICT

Input:
Output: Backtracking decision level + a new conflict clause

1. **if** *current-decision-level* = 0 **then return** -1;
2. *cl* := *current-conflicting-clause*;
3. **while** (¬STOP-CRITERION-MET(*cl*)) **do**
4. *lit* := LAST-ASSIGNED-LITERAL(*cl*);
5. *var* := VARIABLE-OF-LITERAL(*lit*);
6. *ante* := ANTECEDENT(*lit*);
7. *cl* := RESOLVE(*cl*, *ante*, *var*);
8. add-clause-to-database(*cl*);
9. **return** clause-asserting-level(*cl*); ▷ 2nd highest decision level in *cl*

Definition 2.9 (unique implication point (UIP)). *Given a partial conflict graph corresponding to the decision level of the conflict, a* unique implication point (UIP) *is any node other than the conflict node that is on all paths from the decision node to the conflict node.*

The decision node itself is a UIP by definition, while other UIPs, if they exist, are internal nodes corresponding to implications at the decision level of the conflict. In graph-theoretical terms, UIPs *dominate* the conflict node.

Definition 2.10 (first UIP). *A* first UIP *is a UIP that is closest to the conflict node.*

We leave the proof that the notion of a first UIP in a conflict graph is well defined as an exercise (see Problem 2.14). Figure 2.9 demonstrates UIPs in a conflict graph (see also the caption).

Empirical studies show that a good strategy for STOP-CRITERION-MET(*cl*) (line 3) is to return TRUE if and only if *cl* contains the negation of the first UIP as its single literal at the current decision level. This negated literal becomes asserted immediately after backtracking. There are several advantages to this strategy, which may explain the results of the empirical studies:

1. *The strategy has a low computational cost, compared with strategies that choose UIPs further away from the conflict.*
2. *It backtracks to the lowest decision level.*

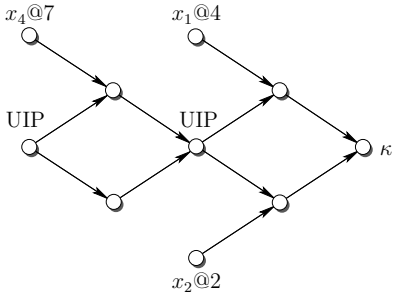

Fig. 2.9. An implication graph (stripped of most of its labels) with two UIPs. The left UIP is the decision node, and the right one is the first UIP, as it is the one closest to the conflict node

The second fact can be demonstrated with the help of Fig. 2.9. Let l_1 and l_2 denote the literals at the first and the second UIP, respectively. The asserting clauses generated with the first-UIP and second-UIP strategies are, respectively, $(\neg l_1 \vee \neg x_1 \vee \neg x_2)$ and $(\neg l_2 \vee \neg x_1 \vee \neg x_2 \vee \neg x_4)$. It is not a coincidence that the second clause subsumes the first, other than the asserting literals $\neg l_1$ and $\neg l_2$: it is always like this, by construction. Now recall how the backtracking level is determined: it is equal to the decision level corresponding to the second highest in the asserting clause. Clearly, this implies that the backtracking level computed with regard to the first clause is lower than that computed with regard to the second clause. In our example, these are decision levels 4 and 7, respectively.

In order to explain lines 4–7 of Analyze-Conflict, we need the following definition:

Definition 2.11 (binary resolution and related terms). *Consider the following inference rule:*

$$\frac{(a_1 \vee \ldots \vee a_n \vee \beta) \qquad (b_1 \vee \ldots \vee b_m \vee \neg\beta)}{(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)} \text{ (Binary Resolution)}, \quad (2.9)$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are literals and β is a variable. The variable β is called the **resolution variable**. *The clauses $(a_1 \vee \ldots \vee a_n \vee \beta)$ and $(b_1 \vee \ldots \vee b_m \vee (\neg\beta))$ are the* **resolving clauses**, *and $(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$ is the* **resolvent clause**.

A well-known result obtained by Robinson [243] shows that a deductive system based on the binary-resolution rule as its single inference rule is sound and complete. In other words, a CNF formula is unsatisfiable if and only if there exists a finite series of binary-resolution steps ending with the empty clause.

The function Resolve(c_1, c_2, v) used in line 7 of Analyze-Conflict returns the resolvent of the clauses c_1, c_2, where the resolution variable is v. The

Aside: Hard Problems for Resolution-Based Procedures
Some propositional formulas can be decided with no less than an exponential number of resolution steps in the size of the input. Haken [137] proved in 1985 that the **pigeonhole problem** is one such problem: given $n > 1$ pigeons and $n - 1$ pigeonholes, can each of the pigeons be assigned a pigeonhole without sharing? While a formulation of this problem in propositional logic is rather trivial with $n \cdot (n-1)$ variables, currently no SAT solver (which, recall, implicitly perform resolution) can solve this problem in a reasonable amount of time for n larger than several tens, although the size of the CNF itself is relatively small. As an experiment, we tried to solve this problem for $n = 20$ with four leading SAT solvers: Siege4 [248], zChaff-04 [202], HaifaSat [124], and Glucose-2014 [8]. On a Pentium 4 with 1 GB of main memory, none of them could solve this problem within three hours. Compare this result with the fact that, bounded by the same timeout, these tools routinely solve problems arising in industry with hundreds of thousands and even millions of variables.

The good news is that some SAT solvers now support a preprocessing step with which **cardinality constraints** (constraints of the form $\Sigma_i x_i \leq k$) are identified in the CNF, and solved by a separate technique. The pigeonhole problem implicitly uses a cardinality constraint of the form $\Sigma_i x_i \leq 1$ for each pigeonhole, to encode the fact that it can hold at most one pigeon, and indeed a SAT solver such as SAT4J, which supports this technique, can solve this problem even with $n = 200$ [32].

Antecedent function used in line 6 of this function returns $Antecedent(lit)$. The other functions and variables are self-explanatory.

Analyze-Conflict progresses from right to left on the conflict graph, starting from the conflicting clause, while constructing the new conflict clause through a series of resolution steps. It begins with the conflicting clause cl, in which all literals are set to 0. The literal lit is the literal in cl assigned last, and var denotes its associated variable. The antecedent clause of var, denoted by $ante$, contains $\neg lit$ as the only satisfied literal, and other literals, all of which are currently unsatisfied. The clauses cl and $ante$ thus contain lit and $\neg lit$, respectively, and can therefore be resolved with the resolution variable var. The resolvent clause is again a conflicting clause, which is the basis for the next resolution step.

Example 2.12. Consider the partial implication graph and set of clauses in Fig. 2.10, and assume that the implication order in the BCP was x_4, x_5, x_6, x_7.

The conflict clause $c_5 := (x_{10} \vee x_2 \vee \neg x_4)$ is computed through a series of binary resolutions. Analyze-Conflict traverses backwards through the implication graph starting from the conflicting clause c_4, while following the order of the implications in reverse, as can be seen in the table below. The

$c_1 = (\neg x_4 \vee x_2 \vee x_5)$
$c_2 = (\neg x_4 \vee x_{10} \vee x_6)$
$c_3 = (\neg x_5 \vee \neg x_6 \vee \neg x_7)$
$c_4 = (\neg x_6 \vee x_7)$
$\vdots$

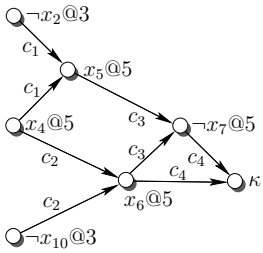

Fig. 2.10. A partial implication graph and a set of clauses that demonstrate Algorithm 2.2.2. The nodes are depicted so that their horizontal position is consistent with the order in which they were created. Algorithm 2.2.2 traverses the nodes in reverse order, from right to left. The first UIP it finds is x_4, and, correspondingly, the asserted literal is $\neg x_4$

intermediate clauses, in this case the second and third clauses in the resolution sequence, are typically discarded.

Name	*cl*	*lit*	*var*	*ante*
c_4	$(\neg x_6 \vee x_7)$	x_7	x_7	c_3
	$(\neg x_5 \vee \neg x_6)$	$\neg x_6$	x_6	c_2
	$(\neg x_4 \vee x_{10} \vee \neg x_5)$	$\neg x_5$	x_5	c_1
c_5	$(\neg x_4 \vee x_2 \vee x_{10})$			

The clause c_5 is an asserting clause in which the negation of the first UIP (x_4) is the only literal from the current decision level. ▟

2.2.5 Decision Heuristics

Probably the most important element in SAT solving is the strategy by which the variables and the value given to them are chosen. This strategy is called the **decision heuristic** of the SAT solver. Let us survey some of the best-known decision heuristics, in the order in which they were suggested, which is also the order of their average efficiency as measured by numerous experiments. New strategies are published every year.

Jeroslow–Wang

Given a CNF formula $\mathcal{B}$, compute for each literal l

$$J(l) = \Sigma_{\omega \in \mathcal{B}, l \in \omega} 2^{-|\omega|} \ , \tag{2.10}$$

where ω represents a clause and $|\omega|$ its length. Choose the literal l for which $J(l)$ is maximal, and for which neither l or $\neg l$ is asserted.

This strategy gives higher priority to literals that appear frequently in short clauses. It can be implemented statically (one computation at the beginning of the run) or dynamically, where in each decision only unsatisfied clauses are considered in the computation. The dynamic approach produces better decisions, but also imposes large overhead at each decision point.

Dynamic Largest Individual Sum (DLIS)

At each decision level, choose the unassigned literal that satisfies the largest number of currently unsatisfied clauses.

The common way to implement such a heuristic is to keep a pointer from each literal to a list of clauses in which it appears. At each decision level, the solver counts the number of clauses that include this literal and are not yet satisfied, and assigns this number to the literal. Subsequently, the literal with the largest count is chosen. DLIS imposes a large overhead, since the complexity of making a decision is proportional to the number of clauses. Another variation of this strategy, suggested by Copty et al. [79], is to count the number of satisfied clauses resulting from each possible decision *and its implications through BCP*. This variation indeed makes better decisions, but also imposes more overhead.

Variable State Independent Decaying Sum (VSIDS) and Variants

This is a strategy that was introduced in the SAT solver Chaff [202], which is reminiscent of DLIS but far more efficient. First, when counting the number of clauses in which every literal appears, disregard the question of whether that clause is already satisfied or not. This means that the estimation of the quality of every decision is compromised, but the complexity of making a decision is better: it takes a constant time to make a decision assuming we keep the literals in a list sorted by their score. Second, periodically divide all scores by 2.

The idea is to make the decision heuristic **conflict-driven**, which means that it tries to solve recently discovered conflicts first. For this purpose, it needs to give higher scores to variables that are involved in recent conflicts. Recall that every conflict results in a conflict clause. A new conflict clause, like any other clause, adds 1 to the score of each literal that appears in it. The greater the amount of time that has passed since this clause was added, the more often the score of these literals is divided by 2. Thus, variables in new conflict clauses become more influential. The SAT solver Chaff, which introduced VSIDS, allows one to tune this strategy by controlling the frequency with which the scores are divided and the constant by which they are divided. It turns out that different families of CNF formulas are best solved with different parameters.

There are other conflict-driven heuristics. Consider, for example, the strategy adopted by the award-winning SAT solver MiniSAT. MiniSAT maintains

an activity score for each variable (in the form of a floating-point number with double precision), which measures the involvement of each variable in inferring new clauses. If a clause c is inferred from clauses $c_1, \ldots, c_n$, then each instance of a variable v in $c_1, \ldots, c_n$ entails an increase in the score of v by some constant *inc*. *inc* is initially set to 1, and then multiplied by 1.05 after each conflict, thus giving higher score to variables that participate in recent conflicts. To prevent overflow, if the activity score of some variable is higher than 10^{100}, then all variable scores as well as *inc* are multiplied by 10^{-100}. The variable that has the highest score is selected. The value chosen for this variable is either FALSE, random, or, when relevant, the previous value that this variable was assigned. The fact that in such a successful solver as MINISAT there is no attempt to guess the right value of a variable indicates that what matters is the locality of the search coupled with learning, rather than a correct guess of the branch. This is not surprising: most branches, even in satisfiable formulas, do not lead to a satisfying assignment.

Clause-Based Heuristics

In this family of heuristics, literals in recent conflict clauses are given absolute priority. This effect is achieved by traversing backwards the list of learned clauses each time a decision has to be made. We begin by describing in detail a heuristic called Berkmin.

Maintain a score per variable, similar to the score VSIDS maintains for each literal (i.e., increase the counter of a variable if one of its literals appears in a clause, and periodically divide the counters by a constant). Maintain a similar score for each literal, but do not divide it periodically. Push conflict clauses into a stack. When a decision has to be made, search for the topmost clause on this stack that is unresolved. From this clause, choose the unassigned variable with the highest variable score. Determine the value of this variable by choosing the literal corresponding to this variable with the highest literal score. If the stack is empty, the same strategy is applied, except that the variable is chosen from the set of all unassigned variables rather than from a single clause.

This heuristic was first implemented in a SAT solver called BERKMIN. The idea is to give variables that appear in recent conflicts absolute priority, which seems empirically to be more effective. It also concentrates only on unresolved conflicts, in contrast to VSIDS.

A different clause-based strategy is called Clause-Move-To-Front (CMTF). It is similar to Berkmin, with the difference that, at the time of learning a new clause, k clauses (k being a constant that can be tuned) that participated in resolving the new clause are pushed to the end of the list, just before the new clause. The justification for this strategy is that it keeps the search more focused. Suppose, for example, that a clause c is resolved from c_1, c_2, and c_3. We can write this as $c_1 \wedge c_2 \wedge c_3 \implies c$, which makes it clear that satisfying c is easier than satisfying $c_1 \wedge c_2 \wedge c_3$. Hence the current partial assignment

contradicted $c_1 \wedge c_2 \wedge c_3$, the solver backtracked, and now tries to satisfy an easier formula, namely c, before returning to those three clauses. CMTF is implemented in a SAT solver called HaifaSat [124], and a variation of it called Clause-Based Heuristic (CBH) is implemented in Eureka [99].

2.2.6 The Resolution Graph and the Unsatisfiable Core

Since each conflict clause is derived from a set of other clauses, we can keep track of this process with a **resolution graph**.

Definition 2.13 (binary resolution graph). *A* binary resolution graph *is a directed acyclic graph where each node is labeled with a clause, each root corresponds to an original clause, and each nonroot node has exactly two incoming edges and corresponds to a clause derived by binary resolution from its parents in the graph.*

Typically, SAT solvers do not retain all the intermediate clauses that are created during the resolution process of the conflict clause. They store enough clauses, however, for building a graph that describes the relation between the conflict clauses.

Definition 2.14 (hyper-resolution graph). *A* hyper-resolution graph *is a directed acyclic graph where each node is labeled with a clause, each root corresponds to an original clause, and each nonroot node has two or more incoming edges and corresponds to a clause derived by binary resolution from its parents in the graph, possibly through other clauses that are not represented in the graph.*

Example 2.15. Consider once again the implication graph in Fig. 2.10. The clauses $c_1, \ldots, c_4$ participate in the resolution of c_5. The corresponding resolution graph appears in Fig. 2.11. ⌟

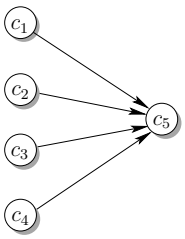

Fig. 2.11. A hyper-resolution graph corresponding to the implication graph in Fig. 2.10

In the case of an unsatisfiable formula, the resolution graph has a sink node (i.e., a node with incoming edges only), which corresponds to an empty clause.[3]

The resolution graph can be used for various purposes, some of which we mention here. The most common use of this graph is for deriving an *unsatisfiable core* of unsatisfiable formulas.

Definition 2.16 (unsatisfiable core). *An* unsatisfiable core *of a CNF unsatisfiable formula is any unsatisfiable subset of the original set of clauses.*

Unsatisfiable cores which are relatively small subsets of the original set of clauses are useful in various contexts, because they help us to focus on a *cause* of unsatisfiability (there can be multiple unsatisfiable cores not contained in each other, and not even intersecting each other). We leave it to the reader in Problem 2.17 to find an algorithm that computes a core given a resolution graph.

Another common use of a resolution graph is for certifying a SAT solver's conclusion that a formula is unsatisfiable. Unlike the case of satisfiable instances, for which the satisfying assignment is an easy-to-check piece of evidence, checking an unsatisfiability result is harder. Using the resolution graph, however, an independent checker can replay the resolution steps starting from the original clauses until it derives the empty clause. This verification requires time that is linear in the size of the resolution proof.

2.2.7 Incremental Satisfiability

In numerous industrial applications the SAT solver is a component in a bigger system that sends it satisfiability queries. For example, a program that plans a path for a robot may use a SAT solver to find out if there exists a path within k steps from the current state. If the answer is negative, it increases k and tries again. The important point here is that the sequence of formulas that the SAT solver is asked to solve is not arbitrary: these formulas have a lot in common. Can we use this fact to make the SAT solver run faster? We should somehow reuse information that was gathered in previous instances to expedite the solution of the current one. To make things simpler, consider two CNF formulas, C_1 and C_2, which are solved consecutively, and assume that C_2 is known at the time of solving C_1. Here are two kinds of information that can be reused when solving C_2:

- *Reuse clauses.* We should answer the following question: if c is a conflict clause learned while solving C_1, under what conditions are C_2 and $C_2 \wedge c$ equisatisfiable? It is easier to answer this question if we view C_1 and C_2

[3] In practice, SAT solvers terminate before they actually derive the empty clause, as can be seen in Algorithms 2.2.1 and 2.2.2, but it is possible to continue developing the resolution graph after the run is over and derive a full resolution proof ending with the empty clause.

as sets of clauses. Let C denote the clauses in the intersection $C_1 \cap C_2$. Any clause learnt solely from C clauses can be reused when solving C_2. In practice, as in the path planning problem mentioned above, consecutive formulas in the sequence are very similar, and hence C_1 and C_2 share the vast majority of their clauses, which means that most of what was learnt can be reused. We leave it as an exercise (Problem 2.16) to design an algorithm that discovers the clauses that can be reused.

- *Reuse heuristic parameters.* Various weights are updated during the solving process, and used to heuristically guide the search, e.g., variable score is used in decision making (Sect. 2.2.5), weights expressing the activity of clauses in deriving new clauses are used for determining which learned clauses should be maintained and which should be deleted, etc. If C_1 and C_2 are sufficiently similar, starting to solve C_2 with the weights at the end of the solving process of C_1 can expedite the solving of C_2.

To understand how modern SAT solvers support incremental solving, one should first understand a mechanism called **assumptions**, which was introduced with the SAT solver MiniSAT [110]. Assumptions are literals that are known to hold when solving C_1, but may be removed or negated when solving C_2. The list of assumption literals is passed to the solver as a parameter. The solver treats assumptions as special literals that dictate the initial set of decisions. If the solver backtracks beyond the decision level of the last assumption, it declares the formula to be unsatisfiable, since there is no solution without changing the assumptions. For example, suppose $a_1, \ldots, a_n$ are the assumption literals. Then the solver begins by making the decisions $a_1 = \text{TRUE}, \ldots, a_n = \text{TRUE}$, while applying BCP as usual. If at any point the solver backtracks to level n or less, it declares the formula to be unsatisfiable.

The key point here is that all clauses that are learnt are *independent of the assumptions* and can therefore be reused when these assumptions no longer hold. This is the nature of learning: it learns clauses that are independent of specific decisions, and assumptions are just decisions. Hence, we can start solving C_2 while maintaining all the clauses that were learnt during the solving process of C_1. Note that in this way we reuse both types of information mentioned above, and save the time of reparsing the formula.

We now describe how assumptions are used for solving the general incremental SAT problem, which requires both addition and deletion of clauses between instances. As for adding clauses, the solver receives the set of clauses that should be added ($C_2 \setminus C_1$ in our case) as part of its interface. Removing clauses is done by adding a new assumption literal (corresponding to a *new* variable) to every clause $c \in (C_1 \setminus C_2)$, negated. For example, if $c = (x_1 \vee x_2)$, then it is replaced with $c' = (\neg a \vee x_1 \vee x_2)$, where a is a new variable. Note that under the assumption $a = \text{TRUE}$, $c = c'$, and hence the added assumption literal does not change the satisfiability of the formula. When solving C_2, however, we replace that assumption with the assumption $a = \text{FALSE}$, which

is equivalent to erasing the clause c. Assumption literals used in this way are called **clause selectors**.

2.2.8 From SAT to the Constraint Satisfaction Problem

In parallel to the research on the SAT problem, there has been a lot of research on the Constraint Satisfaction Problem (CSP) [98], with a lot of cross-fertilization between these two fields. CSP allows arbitrary constraints over variables with finite discrete domains. For example, a CSP instance can be defined by variable domains $x_1 \in \{1 \dots 10\}$, $x_2 \in \{2, 4, 6, \dots, 30\}$, $x_3 \in \{-5, -4, \dots, 5\}$ and a Boolean combination of constraints over these variables

$$\textsc{AllDifferent}(x_1, x_2, x_3) \wedge x_1 < x_3 \; . \tag{2.11}$$

The AllDifferent constraint means that its arguments must be assigned different values. Modern CSP solvers support dozens of such constraints. A propositional formula can be seen as a special case of a CSP model, which does not use constraints, other than the Boolean connectives, and the domains of the variables are limited to $\{0, 1\}$.

CSP solving is an NP problem, and hence can be reduced to SAT in polynomial time.[4] Since the domains are restricted to finite discrete domains, "flattening" them to propositional logic requires some work. For example, a variable with a domain $\{1, \dots, n\}$ can be encoded with $\lceil \log(n) \rceil$ propositional variables. If there are "holes" in the domain, then additional constraints are needed on the values that these variables can be assigned. Similarly, the constraints should be translated to propositional formulas. For example, if x_1 is encoded with propositional variables $b_1, \dots, b_5$ and x_2 with $c_1, \dots c_5$, then the constraint $x_1 \neq x_2$ can be cast in propositional logic as $\vee_{i=1}^{5}(b_i \vee c_i) \wedge (\neg b_i \vee \neg c_i)$, effectively forcing at least one of the bits to be different. AllDifferent is then just a pairwise disequality of all its arguments. Additional bitwise (logarithmic) translation methods appear in Chap. 6, whereas a simpler linear-size translation is the subject of Problem 2.11. Indeed, some of the competitive CSP solvers are just *front-end* utilities that translate the CSP to SAT, either up-front, or lazily. Other solvers handle the constraints directly.

A description of how CSP solvers work is beyond the scope of this book. We will only say that they can be built with a core similar to that of SAT, including decision making, constraint propagation, and learning. Each type of constraint has to be handled separately, however. CSP solvers are typically modular, and hence adding one's favorite constraint to an existing CSP solver is not difficult. Most of the work is in defining a *propagator* for the constraint. A propagator of a constraint c is a function that, given c and the current

[4] This complexity result is based on the set of constraints that is typically supported by CSP solvers. Obviously it is possible to find constraints that will push CSP away from NP.

domains of the variables, can (a) infer reductions in domains that are implied by c, and (b) detect that c is conflicting, i.e., it cannot be satisfied in the current domains. In the example above, a propagator for the $<$ constraint should conclude from $x_1 < x_3$ that the domain of x_1 should be reduced to $\{1, \ldots, 4\}$, because higher values are not *supported* by the current domain of x_3. In other words, for an assignment such as $x_1 = 5$, no value in the current domain of x_3, namely $\{-5, \ldots, 5\}$, can satisfy $x_1 < x_3$. As another example, suppose we have the CSP

$$x_1 \in \{0, 1\}, \quad x_2 \in \{0, 1\}, \quad x_3 \in \{0, 1\}$$
$$\textsc{AllDifferent}(x_1, x_2, x_3) \ .$$

The propagator of AllDifferent should detect that the constraint cannot be satisfied under these domains. The equivalent of a propagator in SAT is BCP—see Sect. 2.2.3. Propagators must be sound, but for some constraints it is too computationally expensive to make them complete (see Definition 1.6). In other words, it may not find all possible domain reductions, and may not always detect a conflict under a partial assignment. This does not change the fact that the overall solver *is* complete, because it *does* detect a conflict with a full assignment.

Given a constraints problem, there are two potential advantages to modeling it as a CSP, rather than in propositional logic:

- CSP as a modeling language is far more readable and succinct, and
- When using a CSP solver that is *not* based on reduction to SAT, one may benefit from the fact that some constraints, such as AllDifferent, have polynomial-time precise propagators, whereas solving the same constraint with SAT after it is reduced to propositional logic is worst-case exponential.[5]

Typically these two potential advantages do not play a major role, however, because (a) most problems that are solved in industry are generated automatically, and hence readability is immaterial, and (b) realistic constraints problems mix many types of constraints, and hence solving them remains exponential. The current view is that neither CSP nor SAT dominate the other in terms of run time, and it is more a question of careful engineering than something substantial.

2.2.9 SAT Solvers: Summary

In this section we have covered the basic elements of modern CDCL solvers, including decision heuristics, learning with conflict clauses, and conflict-driven backtracking. There are various other mechanisms for gaining efficiency that

[5] Certain constraints that have a polynomial propagator can be translated to a specific form of propositional formula that, with the right strategy of the SAT solver, can be solved in polynomial time as well—see [220].

we do not cover in this book, such as efficient implementation of BCP, detection of subsumed clauses, preprocessing and simplification of the formula, deletion of conflict clauses, and **restarts** (i.e., restarting the solver when it seems to be in a hopeless branch of the search tree). The interested reader is referred to the references given in Sect. 2.4.

Let us now reflect on the two approaches to formal reasoning that we described in Sect. 1.1—deduction and enumeration. Can we say that SAT solvers, as described in this section, follow either one of them? On the one hand, SAT solvers can be thought of as searching a binary tree with 2^n leaves, where n is the number of Boolean variables in the input formula. Every leaf is a full assignment, and, hence, traversing all leaves corresponds to enumeration. From this point of view, conflict clauses are generated in order to prune the search space. On the other hand, conflict clauses are *deduced* via the resolution rule from other clauses. If the formula is unsatisfiable, then the sequence of applications of this rule, as listed in the SAT solver's log, is a deductive proof of unsatisfiability. The search heuristic can therefore be understood as a strategy of applying an inference rule—searching for a proof. Thus, the two points of view are equally legitimate.

2.3 Problems

2.3.1 Warm-up Exercises

Problem 2.1 (propositional logic: practice). For each of the following formulas, determine if it is satisfiable, unsatisfiable or valid:

1. $(p \implies (q \implies p))$
2. $(p \wedge q) \wedge (a \implies q) \wedge (b \implies \neg p) \wedge (a \vee b)$
3. $(p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee \neg q) \wedge (\neg p \vee q)$
4. $(a \implies \neg a) \implies \neg a$
5. $(((a \implies p) \wedge (b \implies q)) \wedge (\neg a \vee \neg b)) \implies \neg(p \wedge q)$
6. $(a \wedge b \wedge c) \wedge (a \oplus b \oplus c) \wedge (a \vee b \vee c)$
7. $(a \wedge b \wedge c \wedge d) \wedge (a \oplus b \oplus c \oplus d) \wedge (a \vee b \vee c \vee d)$

where $\oplus$ denotes the XOR operator.

Problem 2.2 (modeling: simple). Consider three persons A, B, and C who need to be seated in a row, but:

- A does not want to sit next to C.
- A does not want to sit in the left chair.
- B does not want to sit to the right of C.

Write a propositional formula that is satisfiable if and only if there is a seat assignment for the three persons that satisfies all constraints. Is the formula satisfiable? If so, give an assignment.

Problem 2.3 (modeling: program equivalence). Show that the two `if-then-else` expressions below are equivalent:

```
!(a || b) ? h : !(a == b) ? f : g          !(!a || !b) ? g : (!a && !b) ? h : f
```

You can assume that the variables have only one bit.

Problem 2.4 (SAT solving). Consider the following set of clauses:

$$\begin{array}{ll} (x_5 \vee \neg x_1 \vee x_3)\,, & (\neg x_1 \vee x_2)\,, \\ (\neg x_2 \vee x_4)\,, & (\neg x_3 \vee \neg x_4)\,, \\ (\neg x_5 \vee x_1)\,, & (\neg x_5 \vee \neg x_6)\,, \\ (x_6 \vee x_1)\,. & \end{array} \tag{2.12}$$

Apply the VSIDS decision heuristic and Analyze-Conflict with conflict-driven backtracking. In the case of a tie (during the application of VSIDS), make a decision that eventually leads to a conflict. Show the implication graph at each decision level.

2.3.2 Propositional Logic

Problem 2.5 (propositional logic: NAND and NOR). Prove that any propositional formula can be equivalently written with

- only a NAND gate,
- only a NOR gate.

2.3.3 Modeling

Problem 2.6 (a reduction from your favorite NP-C problem). Show a reduction to propositional logic from your favorite NP-complete problem (not including SAT itself and problems that appear below). A list of famous NP-complete problems can be found online (some examples are: *vertex-cover*, *hitting-set*, *set-cover*, *knapsack*, *feedback vertex set*, *bin-packing*...). Note that many of those are better known as optimization problems, so begin by formulating a corresponding decision problem. For example, the optimization variant of *vertex-cover* is: find the minimal number of vertices that together touch all edges; the corresponding decision problem is: given a natural number k, is it possible to touch all edges with k vertices?

Problem 2.7 (unwinding a finite automaton). A *nondeterministic finite automaton* is a 5-tuple $\langle Q, \Sigma, \delta, I, F \rangle$, where

- Q is a finite set of states,
- Σ is the alphabet (a finite set of letters),
- $\delta : Q \times \Sigma \mapsto 2^Q$ is the transition function (2^Q is the power set of Q),
- $I \subseteq Q$ is the set of initial states, and

- $F \subseteq Q$ is the set of accepting states.

The transition function determines to which states we can move given the current state and input. The automaton is said to *accept* a finite input string $s_1, \ldots, s_n$ with $s_i \in \Sigma$ if and only if there is a sequence of states $q_0, \ldots, q_n$ with $q_i \in Q$ such that

- $q_0 \in I$,
- $\forall i \in \{1, \ldots, n\}.\ q_i \in \delta(q_{i-1}, s_i)$, and
- $q_n \in F$.

For example, the automaton in Fig. 2.12 is defined by $Q = \{s_1, s_2\}$, $\Sigma = \{a, b\}$, $\delta(s_1, a) = \{s_1\}$, $\delta(s_1, b) = \{s_1, s_2\}$, $I = \{s_1\}$, $F = \{s_2\}$, and accepts strings that end with b. Given a nondeterministic finite automaton $\langle Q, \Sigma, \delta, I, F \rangle$ and a fixed input string $s_1, \ldots, s_n$, $s_i \in \Sigma$, construct a propositional formula that is satisfiable if and only if the automaton accepts the string.

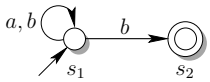

Fig. 2.12. A nondeterministic finite automaton accepting all strings ending with the letter b

Problem 2.8 (assigning teachers to subjects). A problem of covering m subjects with k teachers may be defined as follows. Let $T : \{T_1, \ldots, T_n\}$ be a set of teachers. Let $S : \{S_1, \ldots, S_m\}$ be a set of subjects. Each teacher $t \in T$ can teach some subset $S(t)$ of the subjects S (i.e., $S(t) \subseteq S$). Given a natural number $k \leq n$, is there a subset of size k of the teachers that together covers all m subjects, i.e., a subset $C \subseteq T$ such that $|C| = k$ and $(\bigcup_{t \in C} S(t)) = S$?

Problem 2.9 (Hamiltonian cycle). Show a formulation in propositional logic of the following problem: given a directed graph, does it contain a Hamiltonian cycle (a closed path that visits each node, other than the first, exactly once)?

2.3.4 Complexity

Problem 2.10 (space complexity of CDCL with learning). What is the worst-case space complexity of a CDCL SAT solver as described in Sect. 2.2, in the following cases:

(a) Without learning
(b) With learning, i.e., by recording conflict clauses
(c) With learning in which the length of the recorded conflict clauses is bounded by a natural number k

Problem 2.11 (from CSP to SAT). Suppose we are given a CSP over some unified domain $\mathcal{D} : [min..max]$ where all constraints are of the form $v_i \leq v_j, v_i - v_j \leq c$ or $v_i = v_j + c$ for some constant c. For example

$$((v_2 \leq v_3) \vee (v_4 \leq v_1)) \wedge v_2 = v_1 + 4 \wedge v_4 = v_3 + 3$$

for $v_1, v_2, v_3, v_4 \in [0..7]$ is a formula belonging to this fragment. This formula is satisfied by one of two solutions: $(v_1 \mapsto 0, v_2 \mapsto 4, v_3 \mapsto 4, v_4 \mapsto 7)$, or $(v_3 \mapsto 0, v_1 \mapsto 3, v_4 \mapsto 3, v_2 \mapsto 7)$.

Show a reduction of such formulas to propositional logic. Hint: an encoding which requires $|V| \cdot |\mathcal{D}|$ propositional variables, where $|V|$ is the number of variables and $|\mathcal{D}|$ is the size of the domain, is given by introducing a propositional variable b_{ij} for each variable $v_i \in V$ and $j \in \mathcal{D}$, which indicates that $v_i \leq j$ is true.

For the advanced reader: try to find a logarithmic encoding.

Problem 2.12 (polynomial-time (restricted) SAT). Consider the following two restrictions of CNF:

- A CNF in which there is not more than one positive literal in each clause.
- A CNF formula in which no clause has more than two literals.

1. Show a polynomial-time algorithm that solves each of the problems above.
2. Show that every CNF can be converted to another CNF which is a conjunction of the two types of formula above. In other words, in the resulting formula all the clauses are either unary, binary, or have not more than one positive literal. How many additional variables are necessary for the conversion?

2.3.5 CDCL SAT Solving

Problem 2.13 (backtracking level). We saw that SAT solvers working with conflict-driven backtracking backtrack to the second highest decision level dl in the asserting conflict clause. This wastes all of the work done from decision level $dl + 1$ to the current one, say dl' (although, as we mentioned, this has other advantages that outweigh this drawback). Suppose we try to avoid this waste by performing conflict-driven backtracking as usual, but then repeat the assignments from levels $dl + 1$ to $dl' - 1$ (i.e., override the standard decision heuristic for these decisions). Can it be guaranteed that this reassignment will progress without a conflict?

Problem 2.14 (is the first UIP well defined?). Prove that, in a conflict graph, the notion of a first UIP is well defined, i.e., there is always a single UIP closest to the conflict node. Hint: you may use the notion of *dominators* from graph theory.

2.3.6 Related Problems

Problem 2.15 (blocked clauses). Let φ be a CNF formula, let $c \in \varphi$ be a clause such that $l \in c$ where l is a literal, and let $\varphi_{\neg l} \subseteq \varphi$ be the subset of $\varphi's$ clauses that contain $\neg l$. We say that c is *blocked* by l if the resolution of c with any clause in $\varphi_{\neg l}$ using $var(l)$ as the pivot is a tautology. For example, if $c = (l \vee x \vee y)$ and $\varphi_{\neg l}$ has a single clause $c' = (\neg l \vee \neg x \vee z)$, then resolving c and c' on l results in $(x \vee \neg x \vee y \vee z)$, which is a tautology, and hence c is blocked by l. Prove that φ is equisatisfiable to $\varphi \setminus c$, i.e., blocked clauses can be removed from φ without affecting its satisfiability.

Problem 2.16 (incremental satisfiability). In Sect. 2.2.7 we saw a condition for sharing clauses between similar instances. Suggest a way to implement this check, i.e., how can a SAT solver detect those clauses that were inferred from clauses that are common to both instances? The solution cannot use the mechanism of assumptions.

Problem 2.17 (unsatisfiable cores).

(a) Suggest an algorithm that, given a resolution graph (see Definition 2.14), finds an unsatisfiable core of the original formula that is as small as possible (by this we do not mean that it has to be minimal).
(b) Given an unsatisfiable core, suggest a method that attempts to minimize it further.

Problem 2.18 (unsatisfiable cores and transition clauses). Let $\mathcal{B}$ be an unsatisfiable CNF formula, and let c be a clause of $\mathcal{B}$. If removing c from $\mathcal{B}$ makes $\mathcal{B}$ satisfiable, we say that c is a **transition clause**. Prove the following claim: all transition clauses of a formula are contained in each of its unsatisfiable cores.

2.4 Bibliographic Notes

The very existence of the 980-pages *Handbook of Satisfiability* [35] from 2009, which covers all the topics mentioned in this chapter and much more, indicates what a small fraction of SAT can be covered here. More recently Donald Knuth dedicated almost 300 pages to this topic in his book series *The Art of Computer Programming* [168]. Some highlights from the history of SAT are in order nevertheless.

The Davis–Putnam–Loveland–Logemann (DPLL) framework was a two-stage invention. In 1960, Davis and Putnam considered CNF formulas and offered a procedure to solve them based on an iterative application of three rules [88]: the pure literal rule, the unit clause rule (what we now call BCP), and what they called "the elimination rule", which is a rule for eliminating a variable by invoking resolution (e.g., to eliminate x from a given CNF, apply

resolution to each pair of clauses of the form $(x \vee A) \wedge (\neg x \vee B)$, erase the resolving clauses, and maintain the resolvent). Their motivation was to optimize a previously known incomplete technique for deciding first-order formulas. Note that, at the time, "optimizing" also meant a procedure that was easier to conduct by hand. In 1962, Loveland and Logemann, two programmers hired by Davis and Putnam to implement their idea, concluded that it was more efficient to split and backtrack rather than to apply resolution, and together with Davis published what we know today as the basic DPLL framework [87]. The SAT community tends to distinguish modern solvers from those based on DPLL by referring to them as Conflict-Driven Clause Learning (CDCL) solvers, which emphasizes their learning capability combined with nonchronological backtracking, and the fact that their search is strongly tied to the learning scheme, via a heuristic such as VSIDS. The fact that modern solvers restart the search very frequently adds another major distinction from the earlier DPLL solvers. The main alternative to DPLL/CDCL are the stochastic solvers, also called **local-search** SAT solvers, which were not discussed at length in this chapter. For many years they were led by the GSAT and WalkSat solvers [254]. There are various solvers that combine local search with learning and propagation, such as UnitWalk [143].

The development of SAT solvers has always been influenced by developments in the world of Constraint Satisfaction Problem (CSP), a problem which generalizes SAT to arbitrary finite discrete domains and arbitrary constraints. The definition of CSP by Montanari [201] (and even before that by Waltz in 1975), and the development of efficient CSP solvers, led to cross-fertilization between the two fields: nonchronological backtracking, for example, was first used in CSP, and then adopted by Marques-Silva and Sakallah for their GRASP SAT solver [262], which was the fastest from 1996 to 2000. In addition, learning via conflict clauses in GRASP was inspired by CSP's *nogood recording*. Bayardo and Schrag [21] also published a method for adapting conflict-driven learning to SAT. CSP solvers have an annual competition, called the MiniZinc challenge. The winner of the 2016 competition in the free (unrestricted) search category is Michael Veksler's solver HaifaCSP [279].

The introduction of Chaff in 2001 [202] by Moskewicz, Madigan, Zhao, Zhang, and Malik marked a breakthrough in performance that led to renewed interest in the field. Chaff introduced the idea of conflict-driven nonchronological backtracking coupled with VSIDS, the first conflict-driven decision heuristic. It also included a new mechanism for performing fast BCP based on a data structure called two-watched literals, which is now standard in all competitive solvers. The description of the main SAT procedure in this chapter was inspired mainly by works related to Chaff [298, 299]. Berkmin, a SAT solver developed by Goldberg and Novikov, introduced what we have named "the Berkmin decision heuristic" [133]. The solver Siege introduced Variable-Move-To-Front (VMTF), a decision heuristic that moves a constant number of variables from the conflict clause to the top of the list, which performs very well in practice [248]. MiniSAT [110], a minimalistic open-source solver by

Niklas Eén and Niklas Sörensson, has not only won several competitions in the last decade, but also became a standard platform for SAT research. In the last few competitions there was even a special track for variants of MiniSAT. Starting from 2009, Glucose [8] seems to be one of the leading solvers. It introduced a technique for predicting the quality of a learned clause, based on the number of decision levels that it contains. When the solver erases some of the conflict clauses as most solvers do periodically, this measure improves its chances of keeping those that have a better chance of participating in further learning. The series of solvers by Armin Biere, PicoSAT [30], PrecoSAT, and Lingeling [31] have also been very dominant in the last few years and include dozens of new optimizations. We will only mention here that they are designed for very large CNF instances, and contain accordingly **inprocessing**, i.e., linear-time simplifications of the formula that are done periodically during the search. The simplifications are a selected subset of those that are typically done only as a preprocessing phase. Another very successful technique called **cube-and-conquer** [140] partitions the SAT problem into possibly millions of much easier ones. The challenge is of course to know how to perform this partitioning such that the total run time is reduced. In the first phase, it finds consistent *cubes*, which are simply partial assignments (typically giving values to not more than 10% of the variables); heuristics that are relatively computationally expensive are used in this phase, e.g., checking the impact of each possible assignment on the size of the remaining formula before making the decision. Such an expensive procedure is not competitive if applied throughout the solving process, but here it is used only for identifying relatively short cubes. In the second phase it uses a standard SAT solver to solve the formula after being simplified by the cube. This type of strategy is very suitable for parallelization, and indeed the solver Treengeling, also by Biere, is a highly efficient cube-and-conquer parallel solver. New SAT solvers are introduced every year; readers interested in the latest tools should check the results of the annual SAT competitions.

The realization that different classes of problems are best solved with different solvers led to a strategy of invoking an **algorithm portfolio**. This means that one out of n predefined solvers is chosen automatically for a given problem instance, based on a prediction of which solver is likely to perform best. First, a large "training set" is used for building **empirical hardness models** [212] based on various features of the instances in this set. Then, given a problem instance, the run time of each of the n solvers is predicted, and accordingly the solver is chosen for the task. SATzilla [289] is a successful algorithm portfolio based on these ideas that won several categories in the 2007 competition. Even without the learning phase and the automatic selection of solvers, running different solvers in parallel and reporting the result of the first one to find a solution is a very powerful technique. A parallel version of Lingeling, for example, called Plingeling [31], won the SAT competition in 2010 for parallel solvers, and was much better than any of the single-core ones. It simply runs Lingeling on several cores but each with a different random

seed, and with slightly different parameters that affect its preprocessing and tie-breaking strategies. The various threads only share learned unit clauses.

The connection between the process of deriving conflict clauses and resolution was discussed in, for example, [22, 122, 180, 295, 298]. Zhang and Malik described a procedure for efficient extraction of unsatisfiable cores and unsatisfiability proofs from a SAT solver [298, 299]. There are many algorithms for minimizing such cores—see, for example, [123, 150, 184, 214]. A variant of the minimal unsatisfiable core (MUC) problem is called the **high-level minimal unsatisfiable core** (HLMUC) problem [203, 249]. The input to the problem, in addition to the CNF $\mathcal{B}$, is a set of sets of clauses from $\mathcal{B}$. Rather than minimizing the core, here the problem is to minimize the number of such sets that have a nonempty intersection with the core. This problem has various applications in formal verification as described in the above references.

Incremental satisfiability in its modern version, i.e., the problem of which conflict clauses can be reused when solving a related problem (see Problem 2.16), was introduced by Strichman in [260, 261] and independently by Whittemore, Kim, and Sakallah in [281]. Earlier versions of this problem were more restricted, for example, the work of Hooker [148] and of Kim, Whittemore, Marques-Silva, and Sakallah [164].

There is a very large body of theoretical work on SAT as well. Some examples are: in complexity, SAT was the problem that was used for establishing the NP-complete complexity class by Cook in 1971 [77]; in proof complexity, there is a large body of work on various characteristics of resolution proofs [23] and various restrictions and extensions thereof. In statistical mechanics, physicists study the nature of random formulas [67, 197, 48]: for a given number of variables n, and a given fixed clause size k, a clause is randomly generated by choosing, at uniform, from the $\binom{n}{k} \cdot 2^k$ options. A formula φ is a conjunction of $\alpha \cdot n$ random clauses. It is clear that when $\alpha \to \infty$, φ is unsatisfiable, and when $\alpha = 0$, it is satisfiable. At what value of α is the probability of φ being satisfiable 0.5? The answer is $\alpha = 4.267$. It turns out that formulas constructed with this ratio tend to be the hardest to solve empirically. Furthermore, the larger n is, the sharper the phase transition between SAT and UNSAT, asymptotically reaching a step function, i.e., all formulas with $\alpha > 4.267$ are unsatisfiable, whereas all formulas with $\alpha < 4.267$ are satisfiable. This shift from SAT to UNSAT is called a **phase transition**. There have been several articles about this topic in *Science* [166, 196], *Nature* [200], and even *The New York Times* [157].

Let us conclude these notes by mentioning that, in the first edition of this book, this chapter included about 10 pages on Ordered Binary Decision Diagrams (OBDDs) (frequently called BDDs for short). BDDs were invented by Randal Bryant [55] and were extremely influential in various fields in computer science, most notably in automated theorem proving, symbolic model checking, and other subfields of formal verification. With BDDs one can represent and manipulate propositional formulas. A key benefit of BDDs is the fact that they are *canonical* as long as the BDDs are built following the

same variable order, which means that logically equivalent propositional formulas have identical BDDs. If the BDD is not empty, then the formula is trivially satisfiable, which means that once the BDD is built the satisfiability problem can be solved in constant time. The catch is that the construction itself can take exponential space and time, and indeed in practice nowadays CDCL-based SAT is generally better at solving satisfiability problems. Various SAT-based techniques such as Craig interpolants [193] and Property-Directed Reachability [42] mostly replaced BDDs in verification, although BDD-based engines are still used in commercial model checkers. BDDs also find uses in solving other problems, such as precise existential quantification of propositional formulas, *counting* the number of solutions a propositional formula has (an operation that can be done in linear time in the size of the BDD), and more. Knuth dedicated a large chapter to this topic in *The Art of Computer Programming* [167].

2.5 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
$x_i@d$	(SAT) x_i is assigned TRUE at decision level d	33

http://www.springer.com/978-3-662-50496-3

Decision Procedures
An Algorithmic Point of View
Kroening, D.; Strichman, O.
2016, XXI, 356 p. 64 illus., 5 illus. in color., Hardcover
ISBN: 978-3-662-50496-3