2

Decision Procedures for Propositional Logic

2.1 Propositional Logic

We assume that the reader is familiar with propositional logic. The syntax of formulas in propositional logic is defined by the following grammar:

$$\begin{aligned} \textit{formula} &: \textit{formula} \wedge \textit{formula} \mid \neg\textit{formula} \mid (\textit{formula}) \mid \textit{atom} \\ \textit{atom} &: \textit{Boolean-identifier} \mid \textsc{true} \mid \textsc{false} \end{aligned}$$

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

2.1.1 Motivation

Propositional logic is widely used in diverse areas such as database queries, planning problems in artificial intelligence, automated reasoning and circuit design. Here we 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:

- 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();            | if(a) f();
else                         | else
    if(!a) g();              |     if(b) g();
    else f();                |     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.

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

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:

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

▟

2.2 SAT Solvers

2.2.1 The Progress of SAT Solving

Given a Boolean 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. Figure 2.3 shows a sketch of the progress of these tools through the years. 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.

For many years, SAT solvers were better at solving satisfiable instances than unsatisfiable ones. This is not true anymore. 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 right value, simplify the problem immensely.[2] All of these factors contribute to the fast solving of both satisfiable and unsatisfiable instances.

[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.

[2] Specifically, every formula has what is known as **back-door variables** [200], which are variables that, once given the right value, simplify the formula to the point that it is polynomial to solve.

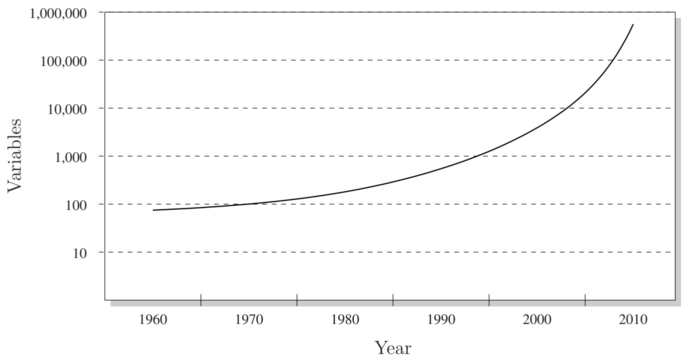


Fig. 2.3. The size of industrial CNF formulas (instances generated for solving various realistic problems such as verification of circuits and planning problems) that are regularly solved by SAT solvers in a few hours, according to year. Most of the progress in efficiency has been made in the last decade

The majority of modern SAT solvers can be classified into two main categories. The first category is based on the *Davis–Putnam–Loveland–Logemann* (**DPLL**) 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, i.e., an assignment to all the variables.

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. DPLL solvers, however, are considered better in most cases, at least at the time of writing this chapter (2007), according to annual competitions that measure their performance with numerous CNF instances. DPLL 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 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 DPLL solvers ineffective. We shall focus on DPLL solvers only.

2.2.2 The DPLL Framework

In its simplest form, a DPLL solver progresses by making a decision about a variable and its value, propagates implications of this decision that are easy to detect, and backtracks 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 in 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 definition 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 $Antecedent(l)$. If more than one unit clause implies l, we refer to the clause that the SAT solver used in order to imply l as its 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 $Antecedent(x_3) = C$. ▟

A framework followed by most modern DPLL solvers has been presented by, for example, Zhang and Malik [211], and is shown in Algorithm 2.2.1. The table in Fig. 2.5 includes a description of the main components used in this algorithm, and Fig. 2.4 depicts the interaction between them. A description of the Analyze-Conflict function is delayed to Sect. 2.2.6.

Algorithm 2.2.1: DPLL-SAT

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

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

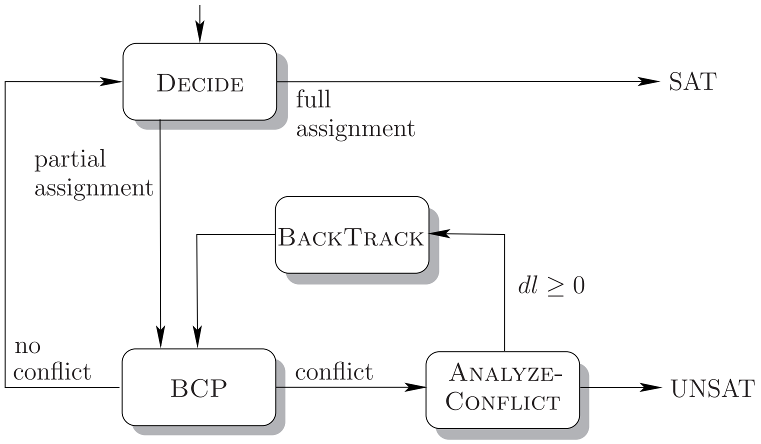


Fig. 2.4. DPLL-SAT: high-level overview of the Davis-Putnam-Loveland-Logemann algorithm. The variable *dl* is the decision level to which the procedure backtracks

2.2.3 BCP and the Implication Graph

$\boxed{x_i@dl}$

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

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 in line 2 because unary clauses at this stage are unit 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. 2.5. A description of the main components of Algorithm 2.2.1

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:*

- *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 [151] – this order is not arbitrary; rather, it is biased towards reaching a conflict faster.

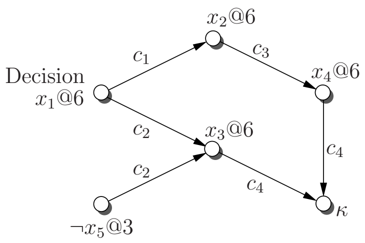
A **partial implication graph** is a subgraph of an implication graph, which illustrates the BCP at a specific decision level. Partial implication graphs are sufficient for describing Analyze-Conflict. The roots in such a partial graph represent assignments (not necessarily decisions) at decision levels 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 $x_1 = 1$. 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. 2.6 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



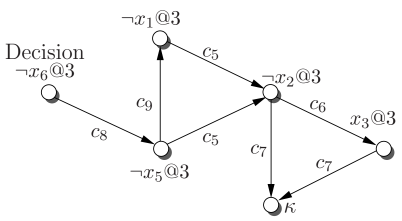


Fig. 2.6. A partial implication graph for decision level 6, corresponding to the clauses in (2.7), after a decision $x_1 = 1$ (*left*) and a similar graph after learning the conflict clause $c_9 = (x_5 \lor \neg x_1)$ and backtracking to decision level 3 (*right*)

for creating this conflict. Therefore, we can safely add to our formula the **conflict clause**

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

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 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*[3] (or, equivalently, it is set to the highest of the decision levels in the clause other than the current decision level), while erasing all decisions and implications made *after* that level.

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 re-starts the BCP process at level 3. Clause c_9 is a special kind of a 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.

[3] In the case of learning a unary clause, the solver backtracks to the ground level.

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.

After asserting $x_1 = 0$ the solver again reaches a conflict, as can be seen in the right drawing in Fig. 2.6. This time the conflict clause (x_2) is added, 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. 2.6 (right).

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) has arrived may skew the search to areas not considered best anymore by the heuristic. There has been some success in overcoming this problem by repeating previous assignments – see [150].
- *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. 2.7. More details of this graph are explained in the caption.

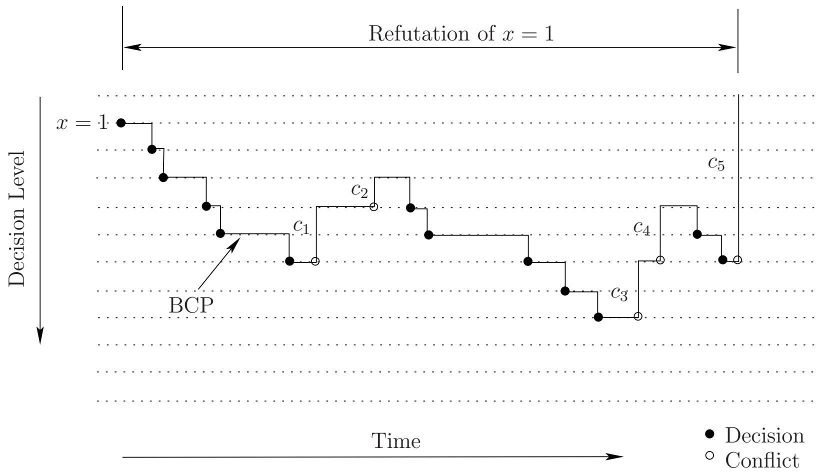


Fig. 2.7. Illustration of the progress of a SAT solver based on conflict-driven backtracking. Every conflict results in a conflict clause (denoted by $c_1, \ldots, c_5$ in the drawing). If the top left decision is $x = 1$, then this drawing illustrates the work done by the SAT solver to refute this wrong 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 $current\text{-}decision\text{-}level = 0$ then return -1;
2. $cl :=$ $current\text{-}conflicting\text{-}clause$;
3. while ($\neg$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.

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 2.11). Figure 2.8 demonstrates UIPs in a conflict graph (see also the caption).

Empirical studies show that a good strategy for the STOP-CRITERION-MET(cl) function (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 to strategies that choose UIPs further away from the conflict.*

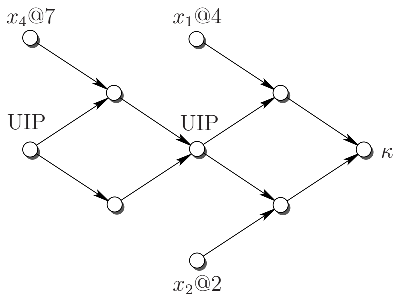


Fig. 2.8. An implication graph (stripped of most of its labels) with two UIPs. The left UIP is the decision node, and the right one is the first UIP, as it is the one closest to the conflict node

2. *It backtracks to the lowest decision level.*

The second fact can be demonstrated with the help of Fig. 2.8. 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:*

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

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 [166] 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.

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 [90] 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 three leading SAT solvers: Siege4 [171], zChaff-04 [133] and HaifaSat [82]. On a Pentium 4 with 1 GB of main memory, none of the three 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 of variables.

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

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

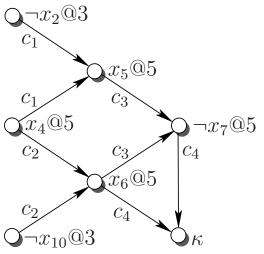


Fig. 2.9. A partial implication graph and a set of clauses that demonstrate Algorithm 2.2.2. The first UIP is x_4, and, correspondingly, the asserted literal is $\neg x_4$

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 in the beginning of the run) or dynamically, where in each decision only unsatisfied clauses are considered in the computation. In the context of a SAT solver that learns through addition of conflict clauses, the dynamic approach is more reasonable.

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. [52], 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)

This is a strategy similar to DLIS, with two differences. First, when counting the number of clauses in which every literal appears, we 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, we periodically divide all scores by 2.

The idea is to make the decision heuristic **conflict-driven**, which means that it tries to solve conflicts before attempting to satisfy more original clauses. 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.

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.

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 (resolution graph). *A* 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 resolution from its parents in the graph, possibly through other clauses that are not represented in the graph.*

Resolution graphs are also called **hyperresolution graphs**, to emphasize that they are not necessarily binary.

Example 2.15. Consider once again the implication graph in Fig. 2.9. The clauses $c_1, \ldots, c_4$ participate in the resolution of c_5. The corresponding resolution graph appears 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.[4]

[4] 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.

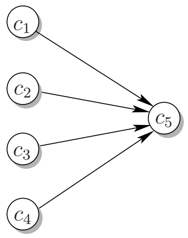


Fig. 2.10. A resolution graph corresponding to the implication graph in Fig. 2.9

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 2.13 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 SAT Solvers: Summary

In this section we have covered the basic elements of modern DPLL solvers, including decision heuristics, learning with conflict clauses, and conflict-driven backtracking. There are various other mechanisms for gaining efficiency that 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. 2.5.

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 legitimate deductive proof of unsatisfiability. The search heuristic can therefore be understood as a strategy of applying an inference rule. Thus, the two points of view are equally legitimate.

2.3 Binary Decision Diagrams

2.3.1 From Binary Decision Trees to ROBDDs

Reduced ordered **binary decision diagrams** (ROBDDs, or **BDD**s for short), are a highly useful graph-based data structure for manipulating Boolean formulas. Unlike CNF, this data representation is **canonical**, which means that if two formulas are equivalent, then their BDD representations are equivalent as well (to achieve this property the two BDDs should be constructed following the same variable order, as we will soon explain). Canonicity is *not* a property of CNF, DNF, or NNF (see Sect. 1.3). Consider, for example, the two CNF formulas

$$\mathcal{B}_1 := \ (x_1 \wedge (x_2 \vee x_3)) \ , \qquad \mathcal{B}_2 := \ (x_1 \wedge (x_1 \vee x_2) \wedge (x_2 \vee x_3)) \ . \tag{2.11}$$

Although the two formulas are in the same normal form and logically equivalent, they are syntactically different. The BDD representations of $\mathcal{B}_1$ and $\mathcal{B}_2$, on the other hand, are the same.

One implication of canonicity is that all tautologies have the same BDD (a single node with a label "1") and all contradictions also have the same BDD (a single node with a label "0"). Thus, although two CNF formulas of completely different size can both be unsatisfiable, their BDD representations are identical: a single node with the label "0". As a consequence, checking for satisfiability, validity, or contradiction can be done in constant time for a given BDD. There is no free lunch, however: building the BDD for a given formula can take exponential space and time, even if in the end it results in a single node.

We start with a simple **binary decision tree** to represent a Boolean formula. Consider the formula

$$\mathcal{B} := \ ((x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3)) \ . \tag{2.12}$$

The binary decision tree in Fig. 2.11 represents this formula with the variable ordering x_1, x_2, x_3. Notice how this order is maintained in each path along the

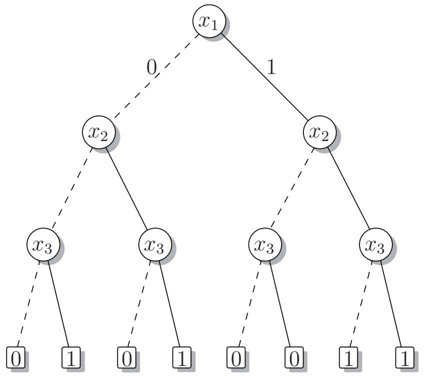


Fig. 2.11. A binary decision tree for (2.12). The drawing follows the convention by which dashed edges represent an assignment of 0 to the variable labeling the source node

tree, and that each of these variables appears exactly once in each path from the root to one of the leaves.

Such a binary decision tree is not any better, in terms of space consumption, than an explicit truth table, as it has 2^n leaves. Every path in this tree, from root to leaf, corresponds to an assignment. Every path that leads to a leaf "1" corresponds to a *satisfying* assignment. For example, the path $x_1 = 1, x_2 = 1, x_3 = 0$ corresponds to a satisfying assignment of our formula $\mathcal{B}$ because it ends in a leaf with the label "1". Altogether, four assignments satisfy this formula. The question is whether we can do better than a binary decision tree in terms of space consumption, as there is obvious redundancy in this tree. We now demonstrate the three **reduction rules** that can be applied to such trees. Together they define what a *reduced ordered BDD* is.

- **Reduction #1.** *Merge the leaf nodes into two nodes* "1" *and* "0". The result of this reduction appears in Fig. 2.12.
- **Reduction #2.** *Merge isomorphic subtrees.* Isomorphic subtrees are subtrees that have roots that represent the same variable (if these are leaves, then they represent the same Boolean value), and have left and right children that are isomorphic as well. After applying this rule to our graph, we are left with the diagram in Fig. 2.13. Note how the subtrees rooted at the left two nodes labeled with x_3 are isomorphic and are therefore merged in this reduction.
- **Reduction #3.** *Removing redundant nodes.* In the diagram in Fig. 2.13, it is clear that the left x_2 node is redundant, because its value does not affect the values of paths that go through it. The same can be said about the middle and right nodes corresponding to x_3. In each such case, we can simply remove the node, while redirecting its incoming edge to the node

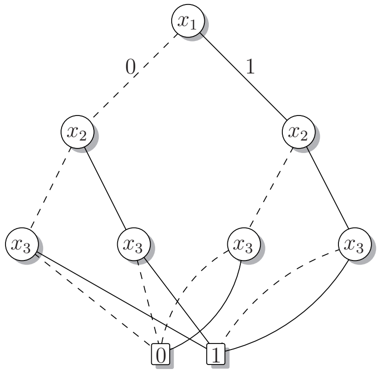


Fig. 2.12. After applying reduction #1, merging the leaf nodes into two nodes

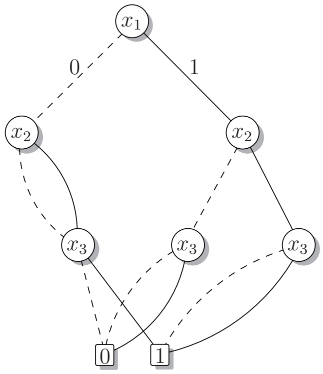


Fig. 2.13. After applying reduction #2, merging isomorphic subtrees

to which both of its edges point. This reduction results in the diagram in Fig. 2.14.

The second and third reductions are repeated as long as they can be applied. At the end of this process, the BDD is said to be *reduced*.

Several important properties of binary trees are maintained during the reduction process:

1. Each terminal node v is associated with a Boolean value $val(v)$. Each nonterminal node v is associated with a variable, denoted by $var(v) \in Var(\mathcal{B})$. $\boxed{val(v)}$ $\boxed{var(v)}$
2. Every nonterminal node v has exactly two children, denoted by $low(v)$ and $high(v)$, corresponding to a FALSE or TRUE assignment to $var(v)$. $\boxed{low(v)}$ $\boxed{high(v)}$
3. Every path from the root to a leaf node contains not more than one occurrence of each variable. Further, the order of variables in each such path is consistent with the order in the original binary tree.

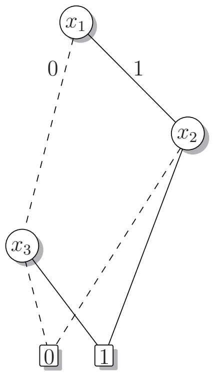


Fig. 2.14. After applying reduction #3, removing redundant nodes

4. A path to the "1" node through all variables corresponds to an assignment that satisfies the formula.

Unlike a binary tree, a BDD can have paths to the leaf nodes through only *some* of the variables. Such paths to the "1" node satisfy the formula regardless of the values given to the other variables, which are appropriately known by the name **don't cares**. A reduced BDD has the property that it does not contain any redundant nodes or isomorphic subtrees, and, as indicated earlier, it is canonical.

2.3.2 Building BDDs from Formulas

The process of turning a binary tree into a BDD helps us to explain the reduction rules, but is not very useful by itself, as we do not want to build the binary decision tree in the first place, owing to its exponential size. Instead, we create the ROBDDs directly: given a formula, we build its BDD recursively from the BDDs of its subexpressions. For this purpose, Bryant defined the procedure APPLY, which, given two BDDs $\mathcal{B}$ and $\mathcal{B}'$, builds a BDD for $\mathcal{B} \star \mathcal{B}'$, where $\star$ stands for any one of the 16 binary Boolean operators (such as "$\wedge$", "$\vee$", and "$\Longrightarrow$"). The complexity of APPLY is bounded by $|\mathcal{B}| \cdot |\mathcal{B}'|$, where $|\mathcal{B}|$ and $|\mathcal{B}'|$ denote the respective sizes of $\mathcal{B}$ and $\mathcal{B}'$.

$\boxed{\mathcal{B} \star \mathcal{B}'}$

In order to describe APPLY, we first need to define the **restrict** operation. This operation is simply an assignment of a value to one of the variables in the BDD. We denote the restriction of $\mathcal{B}$ to $x = 0$ by $\mathcal{B}|_{x=0}$ or, in other words, the BDD corresponding to the function $\mathcal{B}$ after assigning 0 to x. Given the BDD for $\mathcal{B}$, it is straightforward to compute its restriction to $x = 0$. For every node v such that $var(v) = x$, we remove v and redirect the incoming edges of v to $low(v)$. Similarly, if the restriction is $x = 1$, we redirect all the incoming edges to $high(v)$.

$\boxed{\mathcal{B}|_{x=0}}$

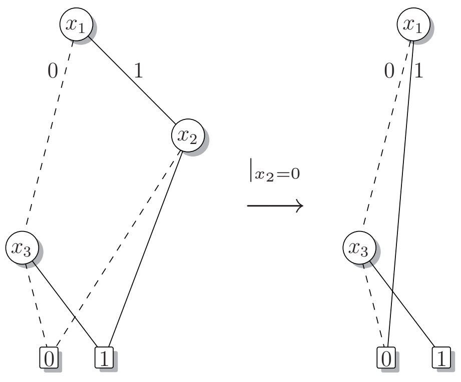


Fig. 2.15. Restricting $\mathcal{B}$ to $x_2 = 0$. This operation is denoted by $\mathcal{B}|_{x_2=0}$

Let $\mathcal{B}$ denote the function represented by the BDD in Fig. 2.14. The diagram in Fig. 2.15 corresponds to $\mathcal{B}|_{x_2=0}$, which is the function $\neg x_1 \wedge x_3$. Let v and v' denote the root variables of $\mathcal{B}$ and $\mathcal{B}'$, respectively, and let $var(v) = x$ and $var(v') = x'$. Apply operates recursively on the BDD structure, following one of these four cases:

1. If v and v' are both terminal nodes, then $\mathcal{B} \star \mathcal{B}'$ is a terminal node with the value $val(v) \star val(v')$.
2. If $x = x'$, that is, the roots of both $\mathcal{B}$ and $\mathcal{B}'$ correspond to the same variable, then we apply what is known as **Shannon expansion**:
$$\mathcal{B} \star \mathcal{B}' := \ (\neg x \wedge (\mathcal{B}|_{x=0} \star \mathcal{B}'|_{x=0})) \vee (x \wedge (\mathcal{B}|_{x=1} \star \mathcal{B}'|_{x=1}))\ . \qquad (2.13)$$
Thus, the resulting BDD has a new node v'' such that $var(v'') = x$, $low(v'')$ points to a BDD representing $\mathcal{B}|_{x=0} \star \mathcal{B}'_{x=0}$, and $high(v'')$ points to a BDD representing $\mathcal{B}|_{x=1} \star \mathcal{B}'|_{x=1}$. Note that both of these restricted BDDs refer to a smaller set of variables than do $\mathcal{B}$ and $\mathcal{B}'$. Therefore, if $\mathcal{B}$ and $\mathcal{B}'$ refer to the same set of variables, then this process eventually reaches the leaves, which are handled by the first case.
3. If $x \neq x'$ and x precedes x' in the given variable order, we again apply Shannon expansion, except that this time we use the fact that the value of x does not affect the value of $\mathcal{B}'$, that is, $\mathcal{B}'|_{x=0} = \mathcal{B}'|_{x=1} = \mathcal{B}'$. Thus, the formula above simplifies to
$$\mathcal{B} \star \mathcal{B}' := \ (\neg x \wedge (\mathcal{B}|_{x=0} \star \mathcal{B}')) \vee (x \wedge (\mathcal{B}|_{x=1} \star \mathcal{B}'))\ . \qquad (2.14)$$
Once again, the resulting BDD has a new node v'' such that $var(v'') = x$, $low(v'')$ points to a BDD representing $\mathcal{B}|_{x=0} \star \mathcal{B}'$, and $high(v'')$ points to a BDD representing $\mathcal{B}|_{x=1} \star \mathcal{B}'$. Thus, the only difference is that we reuse $\mathcal{B}'$ in the recursive call as is, instead of its restriction to $x = 0$ or $x = 1$.
4. The case in which $x \neq x'$ and x follows x' in the given variable order is dual to the previous case.

We now demonstrate Apply with an example.

Example 2.17. Assume that we are given the BDDs for $\mathcal{B} := (x_1 \iff x_2)$ and for $\mathcal{B}' := \neg x_2$, and that we want to compute the BDD for $\mathcal{B} \vee \mathcal{B}'$. Both the source BDDs and the target BDD follow the same order: x_1, x_2. Figure 2.16 presents the BDDs for $\mathcal{B}$ and $\mathcal{B}'$.

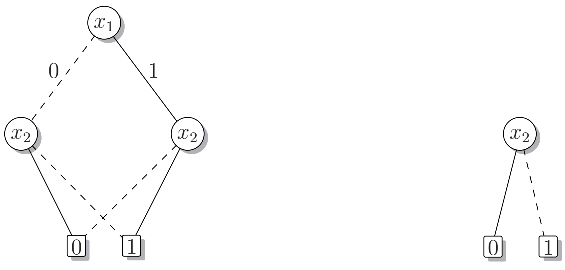


Fig. 2.16. The two BDDs corresponding to $\mathcal{B} := (x_1 \iff x_2)$ (*left*) and $\mathcal{B}' := \neg x_2$ (*right*)

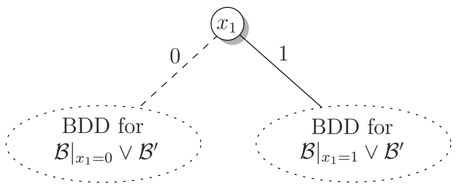

x_1
0
1
BDD for $\mathcal{B}|_{x_1=0} \vee \mathcal{B}'$
BDD for $\mathcal{B}|_{x_1=1} \vee \mathcal{B}'$

Fig. 2.17. Since x_1 appears before x_2 in the variable order, we apply case 3

Since the root nodes of the two BDDs are different, we apply case 3. This results in the diagram in Fig. 2.17. In order to compute the BDD for $\mathcal{B}|_{x_1=0} \vee \mathcal{B}'$, we first compute $\mathcal{B}|_{x_1=0}$. This results in the diagram on the left of Fig. 2.18. To compute $\mathcal{B}|_{x_1=0} \vee \mathcal{B}'$, we apply case 2, as the root nodes refer to the same variable, x_2. This results in the BDD on the right of the figure. Repeating the same process for $high(x_1)$, results in the leaf BDD "1", and thus our final BDD is as shown in Fig. 2.19. This BDD represents the function $x_1 \vee (\neg x_1 \wedge \neg x_2)$, which is indeed the result of $\mathcal{B} \vee \mathcal{B}'$.

▟

The size of the BDD depends strongly on the variable order. That is, constructing the BDD for a given function using different variable orders results in radically different BDDs. There are functions for which one BDD order results in a BDD with a polynomial number of nodes, whereas with a different order the number of nodes is exponential. Bryant gives the function

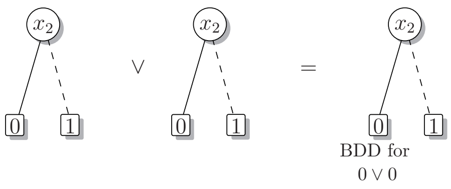


Fig. 2.18. Applying case 2, since the root nodes refer to the same variable. The left and right leaf nodes of the resulting BDD are computed following case 1, since the source nodes are leaves

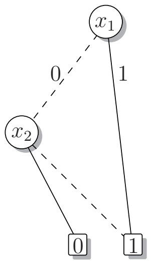


Fig. 2.19. The final BDD for $\mathcal{B} \vee \mathcal{B}'$

$(x_1 \iff x'_1) \wedge \cdots \wedge (x_n \iff x'_n)$ as an example of this phenomenon: using the variable order $x_1, x'_1, x_2, x'_2, \ldots, x_n, x'_n$, the size of the BDD is $3n+2$ while with the order $x_1, x_2, \ldots x_n, x'_1, x'_2, \ldots, x'_n$, the BDD has $3 \cdot 2^n - 1$ nodes. Furthermore, there are functions for which there is no variable order that results in a polynomial number of nodes. Multiplication of bit vectors (arrays of Boolean variables; see Chap. 6) is one such well-known example. Finding a good variable order is a subject that has been researched extensively and has yielded many PhD theses. It is an NP-complete problem to decide whether a given variable order is optimal [36]. Recall that once the BDD has been built, checking satisfiability and validity is a constant-time operation. Thus, if we could always easily find an order in which building the BDD takes polynomial time, this would make satisfiability and validity checking a polynomial-time operation.

There is a very large body of work on BDDs and their extensions – variable-ordering strategies is only one part of this work. Extending BDDs to handle variables of types other than Boolean is an interesting subject, which we briefly discuss as part of Problem 2.15. Another interesting topic is alternatives to Apply. As part of Problem 2.14, we describe one such alternative based on a recursive application of the *ite* (if-then-else) function.

2.4 Problems

2.4.1 Warm-up Exercises

Problem 2.1 (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 2.2 (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 2.3 (SAT solving). Consider the following set of clauses:

$$\begin{array}{ll} (x_5 \vee \neg x_1 \vee x_3)\ , & (\neg x_1 \vee x_2)\ , \\ (\neg x_2 \vee x_4)\ , & (\neg x_3 \vee \neg x_4)\ , \\ (\neg x_5 \vee x_1)\ , & (\neg x_5 \vee \neg x_6)\ , \\ (x_6 \vee x_1)\ . & \end{array} \tag{2.15}$$

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

Problem 2.4 (BDDs). Construct the BDD for $\neg(x_1 \vee (x_2 \wedge \neg x_3))$ with the variable order x_1, x_2, x_3,

(a) starting from a decision tree, and
(b) bottom-up (starting from the BDDs of the atoms x_1, x_2, x_3).

2.4.2 Modeling

Problem 2.5 (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),
- $\delta : Q \times \Sigma \longrightarrow 2^Q$ 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. 2.20 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.

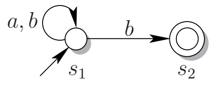


Fig. 2.20. A nondeterministic finite automaton accepting all strings ending with the letter b

Problem 2.6 (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 2.7 (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)?

2.4.3 Complexity

Problem 2.8 (space complexity of DPLL with learning). What is the worst-case space complexity of a DPLL 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 2.9 (polynomial-time (restricted) SAT). Consider the following two restriction 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?

2.4.4 DPLL SAT Solving

Problem 2.10 (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 2.11 (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.

2.4.5 Related Problems

Problem 2.12 (incremental satisfiability). Given two CNF formulas C_1 and C_2, under what conditions can a conflict clause learned while solving C_1 be reused when solving C_2? In other words, if c is a conflict clause learned while solving C_1, under what conditions is C_2 satisfiable if and only if $C_2 \wedge c$ is satisfiable? How can the condition that you suggest be implemented inside a SAT solver? *Hint*: think of CNF formulas as sets of clauses.

Problem 2.13 (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 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.

2.4.6 Binary Decision Diagrams

Problem 2.14 (implementing Apply with *ite*). (Based on [29]) Efficient implementations of BDD packages do not use Apply; rather they use a recursive procedure based on the *ite* (if-then-else) operator. All binary Boolean operators can be expressed as such expressions. For example,

$$\begin{array}{ll} f \vee g = ite(f, 1, g), & f \wedge g = ite(f, g, 0), \\ f \oplus g = ite(f, \neg g, g), & \neg f = ite(f, 0, 1) \; . \end{array} \tag{2.16}$$

How can a BDD for the *ite* operator be constructed? Assume that x labels the root nodes of two BDDs f and g, and that we need to compute $ite(c, f, g)$. Observe the following equivalence:

$$ite(c, f, g) = ite(x, ite(c|_{x=1}, f|_{x=1}, g|_{x=1}), ite(c|_{x=0}, f|_{x=0}, g|_{x=0})) \; . \tag{2.17}$$

Hence, we can construct the BDD for $ite(c, f, g)$ on the basis of a recursive construction. The root node of the result is x, $low(x) = ite(c|_{x=0}, f|_{x=0}, g|_{x=0})$, and $high(x) = ite(c|_{x=1}, f|_{x=1}, g|_{x=1})$. The terminal cases are

$$\begin{array}{l} ite(1, f, g) = ite(0, g, f) = ite(f, 1, 0) = ite(g, f, f) = f \; , \\ ite(f, 0, 1) = \neg f \; . \end{array}$$

1. Let $f := (x \wedge y)$, $g := \neg x$. Show an *ite*-based construction of $f \vee g$.
2. Present pseudocode for constructing a BDD for the *ite* operator. Describe the data structure that you assume. Explain how your algorithm can be used to replace Apply.

Problem 2.15 (binary decision diagrams for non-Boolean functions). (Based on [47].) Let f be a function mapping a vector of m Boolean variables to an integer, i.e., $f : B^m \mapsto \mathbb{Z}$, where $B = \{0, 1\}$.

Let $\{I_1, \ldots, I_N\}$, $N \leq 2^m$, be the set of possible values of f. The function f partitions the space B^m of Boolean vectors into N sets $\{S_1, \ldots, S_N\}$, such that for $i \in \{1 \ldots N\}$, $S_i = \{\bar{x} \mid f(\bar{x}) = I_i\}$ (where $\bar{x}$ denotes a vector). Let f_i be a characteristic function of S_i (i.e., a function mapping a vector $\bar{x}$ to 1 if $f(\bar{x}) \in S_i$ and to 0 otherwise). Every function $f(\bar{x})$ can be rewritten as $\Sigma_{i=1}^{N} f_i(\bar{x}) \cdot I_i$, a form that can be represented as a BDD with $\{I_1, \ldots, I_N\}$ as its terminal nodes. Figure 2.21 shows such a *multiterminal binary decision diagram* (MTBDD) for the function $2x_1 + 2x_2$.

Show an algorithm for computing $f \odot g$, where f and g are multiterminal BDDs, and $\odot$ is some arithmetic binary operation. Compute with your algorithm the MTBDD of $f \odot g$, where

$$\begin{array}{rl} f := & \text{if } x_1 \text{ then } 2x_2 + 1 \text{ else } -x_2 \; , \\ g := & \text{if } x_2 \text{ then } 4x_1 \text{ else } x_3 + 1 \; , \end{array}$$

following the variable order x_1, x_2, x_3.

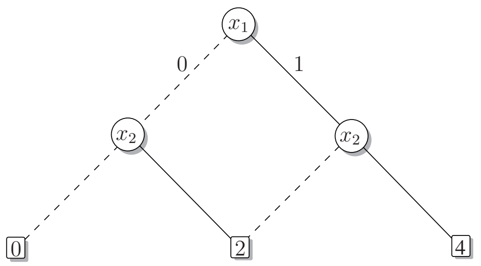


Fig. 2.21. A multiterminal BDD for the function $f(x, y) = 2x_1 + 2x_2$

2.5 Bibliographic Notes

SAT

The Davis–Putnam–Loveland–Logemann framework was a two-stage invention. In 1960, Davis and Putnam considered CNF formulas and offered a procedure to solve it based on an iterative application of three rules [57]: 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 [56]. Numerous SAT solvers were developed through the years on the basis of this framework. The alternative approach of stochastic solvers, which were not discussed in length in this chapter, was led for many years by the GSAT and WalkSat solvers [176].

The definition of the constraints satisfaction problem (CSP) [132] by Montanari (and even before that by Waltz in 1975), a problem which generalizes SAT to arbitrary finite discrete domains and arbitrary constraints, and the development of efficient CSP solvers, led to cross-fertilization between the two fields: nonchronological backtracking, for example, was first used with the CSP, and then adopted by Marques-Silva and Sakallah for their GRASP SAT solver [182], which was the fastest from 1996 to 2000. The addition of conflict clauses in GRASP was also influenced (although in significantly changed form) by earlier techniques called *no-good recording* that were applied to CSP solvers. Bayardo and Schrag [15] also published a method for adapting conflict-driven learning to SAT. The introduction of Chaff in 2001 [133] by Moskewicz, Madigan, Zhao, Zhang and Malik marked a breakthrough in performance that led to renewed interest in the field. These authors introduced the idea of conflict-driven nonchronological backtracking coupled with VSIDS, the first conflict-driven decision heuristic. They also introduced a new mechanism for performing fast BCP, a subject not covered in this chapter, empirically identified the first UIP scheme as the most efficient out of various alternative schemes, and introduced many other means for efficiency. 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 [171]. An indication of how rapid the progress in this field has been was given in the 2006 SAT competition: the best solver in the 2005 competition took ninth place, with a large gap in the run time compared with the 2006 winner, MINISAT-2 [73]. New SAT solvers are introduced every year; readers interested in the latest tools should check the results of the annual SAT competitions. In 2007 the solver RSAT [151] won the "industrial benchmarks" category. RSAT was greatly influenced by MINISAT, but includes various improvements such as ordering of the implications in the BCP stack, an improved policy for restarting the solver, and repeating assignments that are erased while backtracking.

The realization that different classes of problems (e.g., random instances, industrial instances from various problem domains, crafted problems) are best solved with different solvers (or different run time parameters of the same 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** [143] based on various attributes 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 [205] is a successful algorithm portfolio based on these ideas that won several categories in the 2007 competition.

Zhang and Malik described a procedure for efficient extraction of unsatisfiable cores and unsatisfiability proofs from a SAT solver [210, 211]. There are many algorithms for minimizing such cores – see, for example, [81, 98, 118, 144]. The description of the main SAT procedure in this chapter was inspired mainly by [210, 211]. BERKMIN, a SAT solver developed by Goldberg and Novikov, introduced what we have named "the Berkmin decision heuristic" [88]. The connection between the process of deriving conflict clauses and resolution was discussed in, for example, [16, 80, 116, 207, 210].

Incremental satisfiability in its modern version, i.e., the problem of which conflict clauses can be reused when solving a related problem (see Problem 2.12) was introduced by Strichman in [180, 181] and independently by Whittemore, Kim, and Sakallah in [197]. Earlier versions of this problem were more restricted, for example the work of Hooker [96] and of Kim, Whittemore, Marques-Silva, and Sakallah [105].

There is a large body of theoretical work on SAT as well. Probably the best-known is related to complexity theory: SAT played a major role in the theoretical breakthrough achieved by Cook in 1971 [50], who showed that every NP problem can be reduced to SAT. Since SAT is in NP, this made it the first problem to be identified as belonging to the NP-complete complexity class. The general scheme for these reductions (through a translation to a Turing machine) is rarely used and is not efficient. Direct translations of almost all of the well-known NP problems have been suggested through the years, and, indeed, it is always an interesting question whether it is more efficient to solve problems directly or to reduce them to SAT (or to any other NP-complete problem, for that matter). The incredible progress in the efficiency of these solvers in the last decade has made it very appealing to take the translation option. By translating problems to CNF we may lose high-level information about the problem, but we can also gain low-level information that is harder to detect in the original representation of the problem.

An interesting angle of SAT is that it attracts research by physicists![5] Among other questions, they attempt to solve the **phase transition** problem [45, 128]: why and when does a randomly generated SAT problem (according to some well-defined distribution) become hard to solve? There is a well-known result showing empirically that randomly generated SAT instances are hardest when the ratio between the numbers of clauses and variables is around 4.2. A larger ratio makes the formula more likely to be unsatisfiable, and the more constraints there are, the easier it is to detect the unsatisfiability. A lower ratio has the opposite effect: it makes the formula more likely to be satisfiable and easier to solve. Another interesting result is that as the formula grows, the phase transition sharpens, asymptotically reaching a sharp phase transition, i.e., a threshold ratio such that all formulas above it are unsatisfiable, whereas all formulas beneath it are satisfiable. There have been several articles about these topics in *Science* [106, 127], *Nature* [131] and even *The New York Times* [102].

Binary Decision Diagrams

Binary decision diagrams were introduced by Lee in 1959 [115], and explored further by Akers [3]. The full potential for efficient algorithms based on the data structure was investigated by Bryant [35]: his key extensions were to use a fixed variable ordering (for canonical representation) and shared subgraphs (for compression). Together they form what we now refer to as reduced-ordered BDDs. Generally ROBDDs are efficient data structures accompanied by efficient manipulation algorithms for the representation of sets and relations. ROBDDs later became a vital component of symbolic *model checking*, a technique that led to the first large-scale use of formal verification techniques in industry (mainly in the field of electronic design automation). Numerous extensions of ROBDDs exist in the literature, some of which extend

[5] The origin of this interest is in statistical mechanics.

the logic that the data structure can represent beyond propositional logic, and some adapt it to a specific need. Multiterminal BDDs (also discussed in Problem 2.15), for example, were introduced in [47] to obtain efficient *spectral transforms*, and multiplicative binary moment diagrams (*BMDs) [37] were introduced for efficient representation of linear functions. There is also a large body of work on variable ordering in BDDs and dynamic variable reordering (ordering of the variables during the construction of the BDD, rather than according to a predefined list).

It is clear that BDDs can be used everywhere SAT is used (in its basic functionality). SAT is typically more efficient, as it does not require exponential space even in the worst case.[6] The other direction is not as simple, because BDDs, unlike CNF, are canonical. Furthermore, finding all solutions to the Boolean formula represented by a BDD is linear in the number of solutions (all paths leading to the "1" node), while worst-case exponential time is needed for each solution of the CNF. There are various extensions to SAT (algorithms for what is known as **all-SAT**, the problem of finding all solutions to a propositional formula) that attempt to solve this problem in practice using a SAT solver.

2.6 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	30
$val(v)$	(BDD) the 0 or 1 value of a BDD leaf node	45
$var(v)$	(BDD) the variable associated with an internal BDD node	45
$low(v)$	(BDD) the node pointed to by node v when v is assigned 0	45
$high(v)$	(BDD) the node pointed to by node v when v is assigned 1	45
$\mathcal{B} \star \mathcal{B}'$	(BDD) $\star$ is any of the 16 binary Boolean operators	46
$\mathcal{B}\|_{x=0}$	(BDD) simplification of $\mathcal{B}$ after assigning $x = 0$ (also called "restriction")	46

[6] This characteristic of SAT can be achieved by restricting the number of added conflict clauses. In practice, even without this restriction, memory is rarely the bottleneck.