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:

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

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:

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

2.2 SAT Solvers

2.2.1 The Progress of SAT Solving

Given a 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.

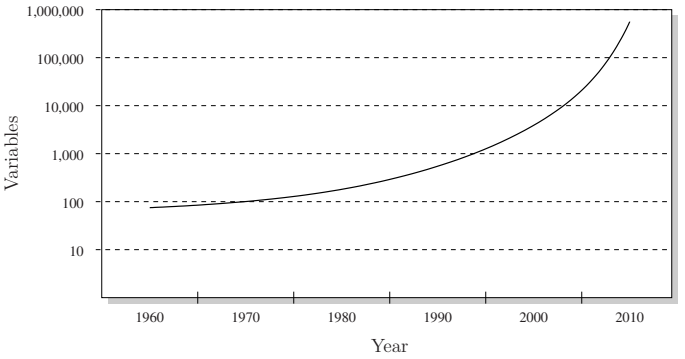

Fig. 2.3. The size of industrial CNF formulas (instances generated for solving various realistic problems such as verification of circuits and planning problems) that are regularly solved by SAT solvers in a few hours, according to year. Most of the progress in efficiency has been made in the last decade

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 $\mathit{Antecedent}(l)$. 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 $\mathit{Antecedent}(x_3) = C$. ⌟

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);
```

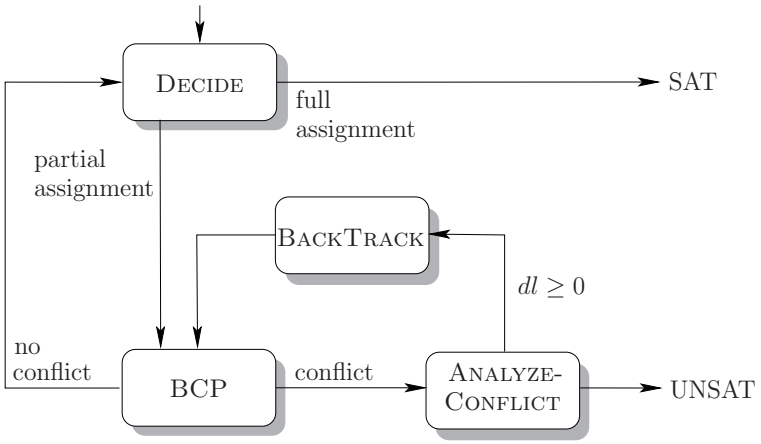

Fig. 2.4. DPLL-SAT: high-level overview of the Davis-Putnam-Loveland-Logemann algorithm. The variable dl is the decision level to which the procedure backtracks

2.2.3 BCP and the Implication Graph

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.

$x_i@dl$

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, $\mathit{Antecedent}(\neg x_5) = c_8$). 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

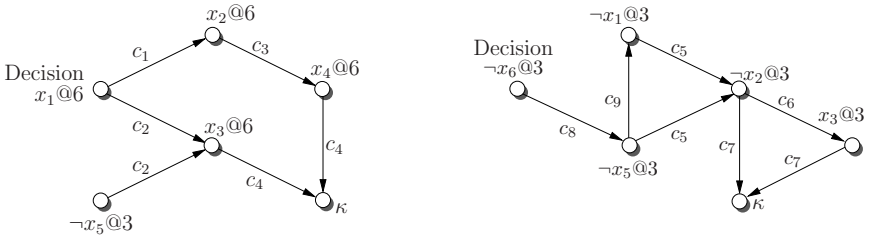

Fig. 2.6. A partial implication graph for decision level 6, corresponding to the clauses in (2.7), after a decision $x_1 = 1$ (*left*) and a similar graph after learning the conflict clause $c_9 = (x_5 \vee \neg x_1)$ and backtracking to decision level 3 (*right*)

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

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

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 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.

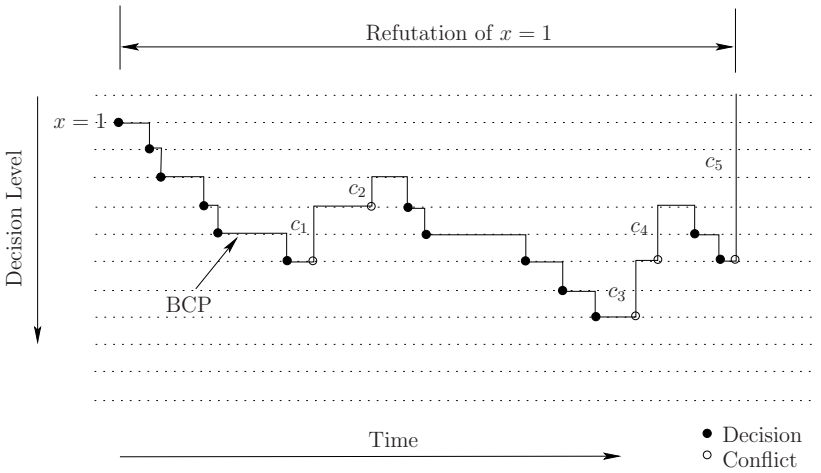

Fig. 2.7. Illustration of the progress of a SAT solver based on conflict-driven backtracking. Every conflict results in a conflict clause (denoted by $c_1, \ldots, c_5$ in the drawing). If the top left decision is $x = 1$, then this drawing illustrates the work done by the SAT solver to refute this wrong 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-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.

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.*

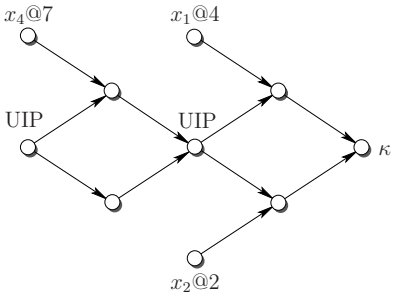

Fig. 2.8. An implication graph (stripped of most of its labels) with two UIPs. The left UIP is the decision node, and the right one is the first UIP, as it is the one closest to the conflict node

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:*

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

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 [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.

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

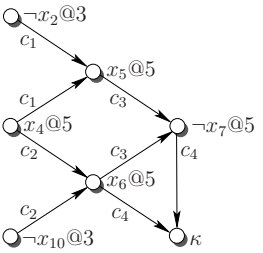

Fig. 2.9. A partial implication graph and a set of clauses that demonstrate Algorithm 2.2.2. The first UIP is x_4, and, correspondingly, the asserted literal is $\neg x_4$

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.

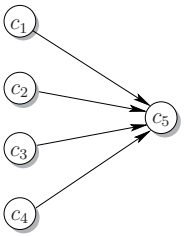

Fig. 2.10. A resolution graph corresponding to the implication graph in Fig. 2.9

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

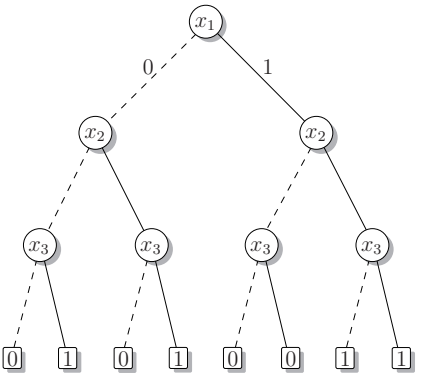

Fig. 2.11. A binary decision tree for (2.12). The drawing follows the convention by which dashed edges represent an assignment of 0 to the variable labeling the source node

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

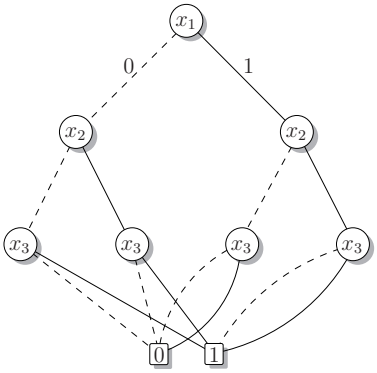

Fig. 2.12. After applying reduction #1, merging the leaf nodes into two nodes

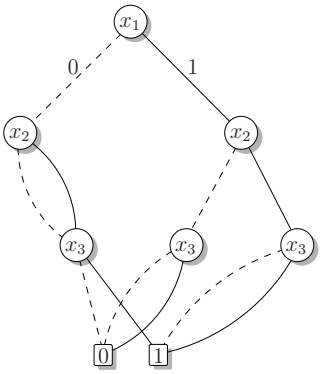

Fig. 2.13. After applying reduction #2, merging isomorphic subtrees

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 $var(v) \in Var(\mathcal{B})$.
2. Every nonterminal node v has exactly two children, denoted by $low(v)$ and $high(v)$, corresponding to a FALSE or TRUE assignment to $var(v)$.
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.

$val(v)$

$var(v)$

$low(v)$

$high(v)$

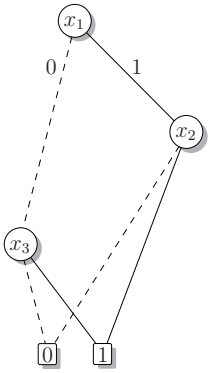

Fig. 2.14. After applying reduction #3, removing redundant nodes

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 $\mathcal{B}'$.

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

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 $high(v)$.

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

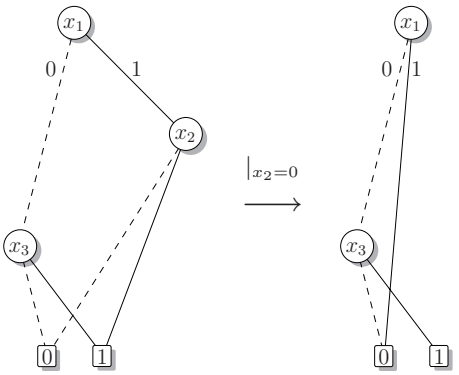

Fig. 2.15. Restricting $\mathcal{B}$ to $x_2 = 0$. This operation is denoted by $\mathcal{B}|_{x_2=0}$

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 $\mathcal{B}'$.

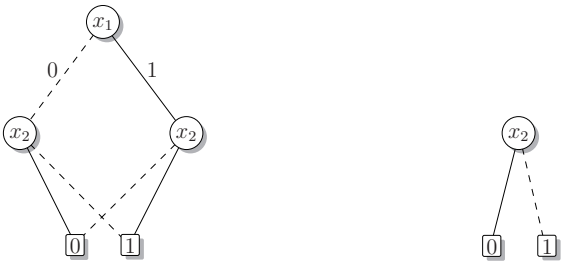

Fig. 2.16. The two BDDs corresponding to $\mathcal{B} := (x_1 \iff x_2)$ (*left*) and $\mathcal{B}' := \neg x_2$ (*right*)

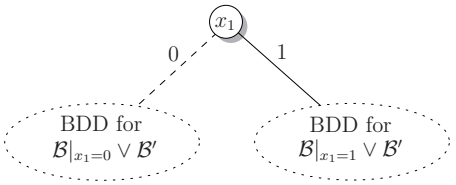

Fig. 2.17. Since x_1 appears before x_2 in the variable order, we apply case 3

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

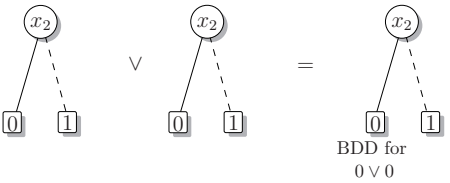

Fig. 2.18. Applying case 2, since the root nodes refer to the same variable. The left and right leaf nodes of the resulting BDD are computed following case 1, since the source nodes are leaves

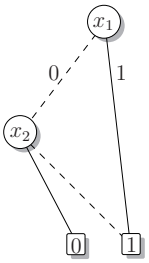

Fig. 2.19. The final BDD for $\mathcal{B} \vee \mathcal{B}'$

$(x_1 \iff x_1') \wedge \cdots \wedge (x_n \iff x_n')$ 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.

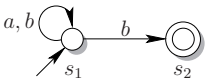

Fig. 2.20. A nondeterministic finite automaton accepting all strings ending with the letter b

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

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

following the variable order x_1, x_2, x_3.

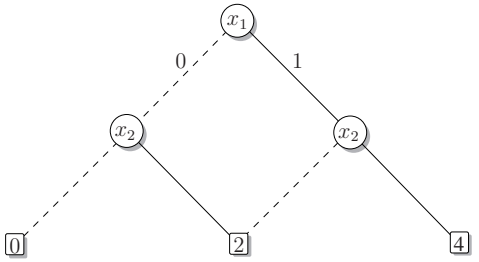

Fig. 2.21. A multiterminal BDD for the function $f(x, y) = 2x_1 + 2x_2$

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.

5

Linear Arithmetic

5.1 Introduction

This chapter introduces decision procedures for conjunctions of linear constraints. An extension of these decision procedures for solving a general linear arithmetic formula, i.e., with an arbitrary Boolean structure, is given in Chap. 11.

Definition 5.1 (linear arithmetic). *The syntax of a formula in* linear arithmetic *is defined by the following rules:*

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\ \mathit{atom} &: \mathit{sum}\ \mathit{op}\ \mathit{sum} \\ \mathit{op} &: = \mid \leq \mid < \\ \mathit{sum} &: \mathit{term} \mid \mathit{sum} + \mathit{term} \\ \mathit{term} &: \mathit{identifier} \mid \mathit{constant} \mid \mathit{constant}\ \mathit{identifier} \end{aligned}$$

The binary minus operator $a - b$ can be read as "syntactic sugar" for $a + -1b$. The operators $\geq$ and $>$ can be replaced by $\leq$ and $<$ if the coefficients are negated. We consider the rational numbers and the integers as domains. For the former domain the problem is polynomial, and for the latter the problem is NP-complete.

As an example, the following is a formula in linear arithmetic:

$$3x_1 + 2x_2 \leq 5x_3 \quad \wedge \quad 2x_1 - 2x_2 = 0\ . \tag{5.1}$$

Note that equality logic, as discussed in Chap. 4, is a fragment of linear arithmetic.

Many problems arising in the code optimization performed by compilers are expressible with linear arithmetic over the integers. As an example, consider the following C code fragment:

```
for(i=1; i<=10; i++)
  a[j+i]=a[j];
```

This fragment is intended to replicate the value of `a[j]` into the locations `a[j+1]` to `a[j+10]`. In a DLX-like assembly language,[1] a compiler might generate the code for the body of the loop as follows. Suppose variable `i` is stored in register `R1`, and variable `j` is stored in register `R2`:

```
R4 ⟵ mem[a+R2]     /* set R4 to a[j] */
R5 ⟵ R2+R1         /* set R5 to j+i */
mem[a+R5] ⟵ R4     /* set a[j+i] to a[j] */
R1 ⟵ R1+1          /* i++ */
```

Code that requires memory access is typically very slow compared with code that operates only on the internal registers of the CPU. Thus, it is highly desirable to avoid load and store instructions. A potential optimization for the code above is to move the load instruction for `a[j]`, i.e., the first statement above, out of the loop body. After this transformation, the load instruction is executed only once at the beginning of the loop, instead of 10 times. However, the correctness of this transformation relies on the fact that the value of `a[j]` does not change within the loop body. We can check this condition by comparing the index of `a[j+i]` with the index of `a[j]` together with the constraint that `i` is between 1 and 10:

$$i \geq 1 \wedge i \leq 10 \wedge j + i = j\,. \tag{5.2}$$

This formula has no satisfying assignment, and thus, the memory accesses cannot overlap. The compiler can safely perform the read access to `a[j]` only once.

5.1.1 Solvers for Linear Arithmetic

The **simplex** method is one of the oldest algorithms for numerical optimization. It is used to find an optimal value for an objective function given a conjunction of linear constraints over real variables. The objective function and the constraints together are called a **linear program** (LP). However, since we are interested in the decision problem rather than the optimization problem, we cover in this chapter a variant of the simplex method called **general simplex** that takes as input a conjunction of linear constraints over the reals *without* an objective function, and decides whether this set is satisfiable.

Integer linear programming, or ILP, is the same problem for constraints over integers. Section 5.3 covers Branch and Bound, an algorithm for deciding such problems.

[1] The DLX architecture is a RISC-like computer architecture, which is similar to the MIPS architecture [149].

These two algorithms can solve conjunctions of a large number of constraints efficiently. We shall also describe two other methods that are considered less efficient, but can still be competitive for solving small problems. We describe them because they are still used in practice, they are relatively easy to implement in their basic form, and they will be mentioned again later in Chap. 11, owing to the fact that they are based on variable elimination. The first of these methods is called **Fourier–Motzkin** variable elimination, and decides the satisfiability of a conjunction of linear constraints over the reals. The second method is called **Omega test**, and decides the satisfiability of a conjunction of linear constraints over the integers.

5.2 The Simplex Algorithm

5.2.1 Decision Problems and Linear Programs

The simplex algorithm, originally developed by Danzig in 1947, decides satisfiability of a conjunction of weak linear inequalities. The set of constraints is normally accompanied by a linear *objective function* in terms of the variables of the formula. If the set of constraints is satisfiable, the simplex algorithm provides a satisfying assignment that maximizes the value of the objective function. Simplex is worst-case exponential. Although there are polynomial-time algorithms for solving this problem (the first known polynomial-time algorithm, introduced by Khachiyan in 1979, is called the **ellipsoid method**), simplex is still considered a very efficient method in practice and the most widely used, apparently because the need for an exponential number of steps is rare in real problems.

As we are concerned with the decision problem rather than the optimization problem, we are going to cover a variant of the simplex algorithm called **general simplex** that does not require an objective function. The general simplex algorithm accepts two types of constraints as input:

1. Equalities of the form

$$a_1 x_1 + \ldots + a_n x_n = 0 \,. \tag{5.3}$$

2. Lower and upper bounds on the variables:[2]

$$l_i \leq x_i \leq u_i \,, \tag{5.4}$$

where l_i and u_i are constants representing the lower and upper bounds on x_i, respectively. The bounds are optional as the algorithm supports unbounded variables.

l_i

u_i

[2] This is in contrast to the classical simplex algorithm, in which all variables are constrained to be nonnegative.

This representation of the input formula is called the **general form**. This statement of the problem does not restrict the modeling power of weak linear constraints, as we can transform an arbitrary weak linear constraint $L \bowtie R$ with $\bowtie \in \{=, \leq, \geq\}$ into the form above as follows. Let m be the number of constraints. For the i-th constraint, $1 \leq i \leq m$:

1. Move all addends in R to the left-hand side to obtain $L' \bowtie b$, where b is a constant.
2. Introduce a new variable s_i. Add the constraints

$$L' - s_i = 0 \quad \text{and} \quad s_i \bowtie b\,. \tag{5.5}$$

If $\bowtie$ is the equality operator, rewrite $s_i = b$ to $s_i \geq b$ and $s_i \leq b$.

The original and the transformed conjunctions of constraints are obviously equisatisfiable.

Example 5.2. Consider the following conjunction of constraints:

$$\begin{aligned} x \;\; +y &\geq 2 \;\wedge \\ 2x \;\; -y &\geq 0 \;\wedge \\ -x \; +2y &\geq 1 \;\; . \end{aligned} \tag{5.6}$$

The problem is rewritten into the general form as follows:

$$\begin{aligned} x \;\; +y \; -s_1 &= 0 \;\; \wedge \\ 2x \;\; -y \; -s_2 &= 0 \;\; \wedge \\ -x \; +2y \; -s_3 &= 0 \;\; \wedge \\ s_1 &\geq 2 \;\; \wedge \\ s_2 &\geq 0 \;\; \wedge \\ s_3 &\geq 1\,. \end{aligned} \tag{5.7}$$

The new variables $s_1, \ldots, s_m$ are called the **additional variables**. The variables $x_1, \ldots, x_n$ in the original constraints are called **problem variables**. Thus, we have n problem variables and m additional variables. As an optimization of the procedure above, an additional variable is only introduced if L' is not already a problem variable or has been assigned an additional variable previously.

5.2.2 Basics of the Simplex Algorithm

It is common and convenient to view linear constraint satisfaction problems as geometrical problems. In geometrical terms, each variable corresponds to a dimension, and each constraint defines a convex subspace: in particular, inequalities define *half-spaces* and equalities define hyperplanes.[3] The (closed)

[3] A hyperplane in a d-dimensional space is a subspace with $d-1$ dimensions. For example, in two dimensions, a hyperplane is a straight line, and in one dimension it is a point.

subspace of satisfying assignments is defined by an intersection of half spaces and hyperplanes, and forms a convex polytope. This is implied by the fact that an intersection between convex subspaces is convex as well. A geometrical representation of the original problem in Example 5.2 appears in Fig. 5.1.

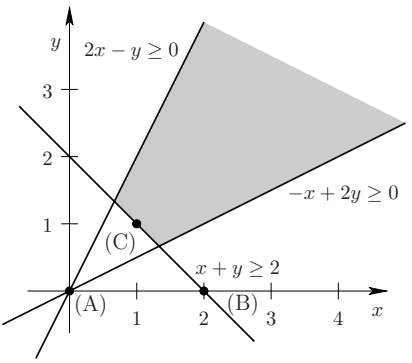

Fig. 5.1. A graphical representation of the problem in Example 5.2, projected on x and y. The shaded region corresponds to the set of satisfying assignments. The marked points (A), (B), and (C) illustrate the progress that the simplex algorithm makes, as will be explained in the rest of this section

It is common to represent the coefficients in the input problem using an m-by-$(n+m)$ matrix A. The variables $x_1, \ldots, x_n, s_1, \ldots, s_m$ are written as a vector $\mathbf{x}$. Following this notation, our problem is equivalent to the existence of a vector $\mathbf{x}$ such that

A
$\mathbf{x}$

$$A\mathbf{x} = 0 \quad \text{and} \quad \bigwedge_{i=1}^{m} l_i \leq s_i \leq u_i \;, \tag{5.8}$$

where $l_i \in \{-\infty\} \cup \mathbb{Q}$ is the lower bound of x_i and $u_i \in \{+\infty\} \cup \mathbb{Q}$ is the upper bound of x_i. The infinity values are for the case that a bound is not set.

Example 5.3. We continue Example 5.2. Using the variable ordering x, y, s_1, s_2, s_3, a matrix representation for the equality constraints in (5.7) is

$$\begin{pmatrix} 1 & 1 & -1 & 0 & 0 \\ 2 & -1 & 0 & -1 & 0 \\ -1 & 2 & 0 & 0 & -1 \end{pmatrix} . \tag{5.9}$$

⌟

Note that a large portion of the matrix in Example 5.3 is very regular: the columns that are added for the additional variables $s_1, \ldots, s_m$ correspond to

an m-by-m diagonal matrix, where the diagonal coefficients are -1. This is a direct consequence of using the general form.

While the matrix A changes as the algorithm progresses, the number of columns of this kind is never reduced. The set of m variables corresponding to these columns are called the **basic variables** and denoted by $\mathcal{B}$. They are also called the *dependent* variables, as their values are determined by those of the nonbasic variables. The nonbasic variables are denoted by $\mathcal{N}$. It is convenient to store and manipulate a representation of A called the **tableau**, which is simply A without the diagonal submatrix. The tableau is thus an m-by-n matrix, where the columns correspond to the nonbasic variables, and each row is associated with a basic variable – the same basic variable that has a "-1" entry at that row in the diagonal sub-matrix in A. Thus, the information originally stored in the diagonal matrix is now represented by the variables labeling the rows.

$\mathcal{B}, \mathcal{N}$

Example 5.4. We continue our running example. The tableau and the bounds for Example 5.2 are:

	x	y
s_1	1	1
s_2	2	-1
s_3	-1	2

$$2 \leq s_1$$
$$0 \leq s_2$$
$$1 \leq s_3$$

The tableau is simply a different representation of A, since $A\mathbf{x} = 0$ can be rewritten into

$$\bigwedge_{x_i \in \mathcal{B}} \left(x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \right). \tag{5.10}$$

When written in the form of a matrix, the sums on the right-hand side of (5.10) correspond exactly to the tableau.

5.2.3 Simplex with Upper and Lower Bounds

The general simplex algorithm maintains, in addition to the tableau, an assignment $\alpha : \mathcal{B} \cup \mathcal{N} \longrightarrow \mathbb{Q}$. The algorithm initializes its data structures as follows:

α

- The set of basic variables $\mathcal{B}$ is the set of additional variables.
- The set of nonbasic variables $\mathcal{N}$ is the set of problem variables.
- For any x_i with $i \in \{1, \ldots, n+m\}$, $\alpha(x_i) = 0$.

If the initial assignment of zero to all variables (i.e., the origin) satisfies all upper and lower bounds of the basic variables, then the formula can be declared satisfiable (recall that initially the nonbasic variables do not have

Algorithm 5.2.1: GENERAL-SIMPLEX

Input: A linear system of constraints S
Output: "Satisfiable" if the system is satisfiable, "Unsatisfiable" otherwise

1. Transform the system into the general form
$$A\mathbf{x} = 0 \quad \text{and} \quad \bigwedge_{i=1}^{m} l_i \leq s_i \leq u_i \ .$$
2. Set $\mathcal{B}$ to be the set of additional variables $s_1, \ldots, s_m$.
3. Construct the tableau for A.
4. Determine a fixed order on the variables.
5. If there is no basic variable that violates its bounds, return "Satisfiable". Otherwise, let x_i be the first basic variable in the order that violates its bounds.
6. Search for the first suitable nonbasic variable x_j in the order for pivoting with x_i. If there is no such variable, return "Unsatisfiable".
7. Perform the pivot operation on x_i and x_j.
8. Go to step 5.

explicit bounds). Otherwise, the algorithm begins a process of changing this assignment.

Algorithm 5.2.1 summarizes the steps of the general simplex procedure. The algorithm maintains two invariants:

- **In-1.** $A\mathbf{x} = 0$
- **In-2.** The values of the nonbasic variables are within their bounds:

$$\forall x_j \in \mathcal{N}. \, l_j \leq \alpha(x_j) \leq u_j \ . \tag{5.11}$$

Clearly, these invariants hold initially since all the variables in $\mathbf{x}$ are set to 0, and the nonbasic variables have no bounds.

The main loop of the algorithm checks if there exists a basic variable that violates its bounds. If there is no such variable, then both the basic and nonbasic variables satisfy their bounds. Owing to invariant **In-1**, this means that the current assignment α satisfies (5.8), and the algorithm returns "Satisfiable".

Otherwise, let x_i be a basic variable that violates its bounds, and assume, without loss of generality, that $\alpha(x_i) > u_i$, i.e., the upper bound of x_i is violated. How do we change the assignment to x_i so it satisfies its bounds? We need to find a way to reduce the value of x_i. Recall how this value is specified:

$$x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \ . \tag{5.12}$$

The value of x_i can be reduced by decreasing the value of a nonbasic variable x_j such that $a_{ij} > 0$ and its current assignment is higher than its lower bound l_j, or by increasing the value of a variable x_j such that $a_{ij} < 0$ and its current assignment is lower than its upper bound u_j. A variable x_j fulfilling one of these conditions is said to be *suitable*. If there are no suitable variables, then the problem is unsatisfiable and the algorithm terminates.

θ Let θ denote by how much we have to increase (or decrease) $\alpha(x_j)$ in order to meet x_i's upper bound:

$$\theta \doteq \frac{u_i - \alpha(x_i)}{a_{ij}} \ . \tag{5.13}$$

Increasing (or decreasing) x_j by θ puts x_i within its bounds. On the other hand x_j does not necessarily satisfy its bounds anymore, and hence may violate the invariant **In-2**. We therefore swap x_i and x_j in the tableau, i.e., make x_i nonbasic and x_j basic. This requires a transformation of the tableau, which is called the **pivot operation**. The pivot operation is repeated until either a satisfying assignment is found, or the system is determined to be unsatisfiable.

The Pivot Operation

Suppose we want to swap x_i with x_j. We will need the following definition:

Definition 5.5 (pivot element, column and row). *Given two variables x_i and x_j, the coefficient a_{ij} is called the* pivot element*. The column of x_j is called the* pivot column*. The row i is called the* pivot row*.*

A precondition for swapping two variables x_i and x_j is that their pivot element is nonzero, i.e., $a_{ij} \neq 0$. The pivot operation (or **pivoting**) is performed as follows:

1. Solve row i for x_j.
2. For all rows $l \neq i$, eliminate x_j by using the equality for x_j obtained from row i.

The reader may observe that the pivot operation is also the basic operation in the well-known **Gaussian variable elimination** procedure.

Example 5.6. We continue our running example. As described above, we initialize $\alpha(x_i) = 0$. This corresponds to point (A) in Fig. 5.1. Recall the tableau and the bounds:

	x	y
s_1	1	1
s_2	2	-1
s_3	-1	2

$$2 \leq s_1$$
$$0 \leq s_2$$
$$1 \leq s_3$$

The lower bound of s_1 is 2, which is violated. The nonbasic variable that is the lowest in the ordering is x. The variable x has a positive coefficient, but no upper bound, and is therefore suitable for the pivot operation. We need to increase s_1 by 2 in order to meet the lower bound, which means that x has to increase by 2 as well ($\theta = 2$). The first step of the pivot operation is to solve the row of s_1 for x:

$$s_1 = x + y \iff x = s_1 - y \,. \tag{5.14}$$

This equality is now used to replace x in the other two rows:

$$s_2 = 2(s_1 - y) - y \iff s_2 = 2s_1 - 3y \tag{5.15}$$

$$s_3 = -(s_1 - y) + 2y \iff s_3 = -s_1 + 3y \tag{5.16}$$

Written as a tableau, the result of the pivot operation is:

	s_1	y
x	1	−1
s_2	2	−3
s_3	−1	3

$$\begin{aligned} \alpha(x) &= 2 \\ \alpha(y) &= 0 \\ \alpha(s_1) &= 2 \\ \alpha(s_2) &= 4 \\ \alpha(s_3) &= -2 \end{aligned}$$

This new state corresponds to point (B) in Fig. 5.1.

The lower bound of s_3 is violated; this is the next basic variable that is selected. The only suitable variable for pivoting is y. We need to add 3 to s_3 in order to meet the lower bound. This translates into

$$\theta = \frac{1 - (-2)}{3} = 1 \,. \tag{5.17}$$

After performing the pivot operation with s_3 and y, the final tableau is:

	s_1	s_3
x	2/3	−1/3
s_2	1	−1
y	1/3	1/3

$$\begin{aligned} \alpha(x) &= 1 \\ \alpha(y) &= 1 \\ \alpha(s_1) &= 2 \\ \alpha(s_2) &= 1 \\ \alpha(s_3) &= 1 \end{aligned}$$

This assignment α satisfies the bounds, and thus $\{x \mapsto 1, y \mapsto 1\}$ is a satisfying assignment. It corresponds to point (C) in Fig. 5.1. ⌟

Selecting the pivot element according to a fixed ordering for the basic and nonbasic variable ensures that no set of basic variables is ever repeated, and hence guarantees termination (no cycling can occur). For a detailed proof see [71]. This way of selecting a pivot element is called **Bland's rule**.

5.2.4 Incremental Problems

Decision problems are often constructed in an **incremental** manner, that is, the formula is strengthened with additional conjuncts. This can make a once satisfiable formula unsatisfiable. One scenario in which an incremental decision procedure is useful is the DPLL(T) framework, which we study in Chap. 11.

The general simplex algorithm is well-suited for incremental problems. First, notice that any constraint can be disabled by removing its corresponding upper and lower bounds. The equality in the tableau is afterwards redundant, but will not render a satisfiable formula unsatisfiable. Second, the pivot operation performed on the tableau is an equivalence transformation, i.e., it preserves the set of solutions. We can therefore start the procedure with the tableau we have obtained from the previous set of bounds.

The addition of upper and lower bounds is implemented as follows:

- If a bound for a nonbasic variable was added, update the values of the nonbasic variables according to the tableau to restore **In-2**.
- Call Algorithm 5.2.1 to determine if the new problem is satisfiable. Start with step 5.

Furthermore, it is often desirable to *remove* constraints after they have been added. This is also relevant in the context of DPLL(T) because this algorithm activates and deactivates constraints. Normally constraints (or rather bounds) are removed when the current set of constraints is unsatisfiable. After removing a constraint the assignment has to be restored to a point at which it satisfied the two invariants of the general simplex algorithm. This can be done by simply restoring the assignment α to the last known satisfying assignment. There is no need to modify the tableau.

5.3 The Branch and Bound Method

Branch and Bound is a widely used method for solving integer linear programs. As in the case of the simplex algorithm, Branch and Bound was developed for solving the optimization problem, but the description here focuses on an adaptation of this algorithm to the decision problem.

The integer linear systems considered here have the same form as described in Sect. 5.2, with the additional requirement that the value of any variable in a satisfying assignment must be drawn from the set of integers. Observe that it is easy to support strict inequalities simply by adding 1 to or subtracting 1 from the constant on the right-hand side.

Definition 5.7 (relaxed problem). *Given an integer linear system S, its* relaxation *is S without the integrality requirement (i.e., the variables are not required to be integer).*

We denote the relaxed problem of S by relaxed(S). Assume the existence of a procedure $LP_{feasible}$, which receives a linear system S as input, and returns "Unsatisfiable" if S is unsatisfiable and a satisfying assignment otherwise. $LP_{feasible}$ can be implemented with, for example, a variation of General-Simplex (Algorithm 5.2.1) that outputs a satisfying assignment if S is satisfiable. Using these notions, Algorithm 5.3.1 decides an integer linear system of constraints (recall that only conjunctions of constraints are considered here).

Algorithm 5.3.1: Feasibility-Branch-and-Bound

Input: An integer linear system S
Output: "Satisfiable" if S is satisfiable, "Unsatisfiable" otherwise

```
1.  procedure Search-integral-solution(S)
2.      res = LP_feasible (relaxed(S));
3.      if res = "Unsatisfiable" then return ;            ▷ prune branch
4.      else
5.          if res is integral then                       ▷ integer solution found
                abort("Satisfiable");
6.          else
7.              Select a variable v that is assigned a nonintegral value r;
8.              Search-integral-solution (S ∪ (v ≤ ⌊r⌋));
9.              Search-integral-solution (S ∪ (v ≥ ⌈r⌉));
10.                                  ▷ no integer solution in this branch


11. procedure Feasibility-Branch-and-Bound(S)
12.     Search-integral-solution(S);
13.     return ("Unsatisfiable");
```

The idea of the algorithm is simple: it solves the relaxed problem with $LP_{feasible}$; if the relaxed problem is unsatisfiable, it backtracks because there is also no integer solution in this branch. If, on the other hand, the relaxed problem is satisfiable and the solution returned by $LP_{feasible}$ happens to be integral, it terminates – a satisfying integral solution has been found. Otherwise, the problem is split into two subproblems, which are then processed with a recursive call. The nature of this split is best illustrated by an example.

Example 5.8. Let $x_1, \ldots, x_4$ be the variables of S. Assume that $LP_{feasible}$ returns the solution

$$(1, 0.7, 2.5, 3) \tag{5.18}$$

in line 2. In line 7, Search-integral-solution chooses between x_2 and x_3, which are the variables that were assigned a nonintegral value. Suppose that

x_2 is chosen. In line 8, S (the linear system solved at the current recursion level) is then augmented with the constraint

$$x_2 \leq 0 \tag{5.19}$$

and sent for solving at a deeper recursion level. If no solution is found in this branch, S is augmented instead with

$$x_2 \geq 1 \tag{5.20}$$

and, once again, is sent to a deeper recursion level. If both these calls return, this implies that S has no satisfying solution, and hence the procedure returns (backtracks). Note that returning from the initial recursion level causes the calling function FEASIBILITY-BRANCH-AND-BOUND to return "Unsatisfiable". ⌟

Algorithm 5.3.1 is not complete: there are cases for which it will branch forever. As noted in [71], the system $1 \leq 3x - 3y \leq 2$, for example, has no integer solutions but unbounded real solutions, and causes the basic Branch and Bound algorithm to loop forever. In order to make the algorithm complete, it is necessary to rely on the small-model property that such formulas have (we used this property earlier in Sect. 4.5). Recall that this means that if there is a satisfying solution, then there is also such a solution within a finite bound, which, for this theory, is also computable. This means that once we have computed this bound on the domain of each variable, we can stop searching for a solution once we have passed it. A detailed study of this bound in the context of optimization problems can be found in [139]. The same bounds are applicable to the feasibility problem as well. Briefly, it was shown in [139] that given an integer linear system S with an $M \times N$ coefficient matrix A, then if there is a solution to S, then one of the extreme points of the convex hull of S is also a solution, and any such solution x^0 is bounded as follows:

$$x_j^0 \leq ((M+N) \cdot N \cdot \theta)^N \quad \text{for } j = 1, \ldots, N\ , \tag{5.21}$$

where θ is the maximal element in the coefficient matrix A or in the vector $\mathbf{b}$. Thus, (5.21) gives us a bound on each of the N variables, which, by adding it as an explicit constraint, forces termination.

Finally, let us mention that BRANCH AND BOUND can be extended in a straightforward way to handle the case in which some of the variables are integers while the others are real. In the context of optimization problems, this problem is known by the name **mixed integer programming**.

5.3.1 Cutting-Planes

Cutting-planes are constraints that are added to a linear system that remove only noninteger solutions; that is, all satisfying integer solutions, if they exist,

Aside: Branch and Bound **for Integer Linear Programs**
When Branch and Bound is used for solving an optimization problem, it becomes somewhat more complicated. In particular, there are various pruning rules based on the value of the current objective function (a branch is pruned if it is identified that it cannot contain a solution better than what is already at hand from another branch). There are also various heuristics for choosing the variable on which to split and the first branch to be explored.

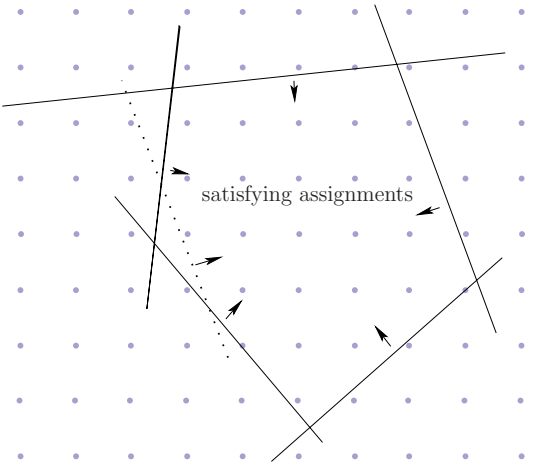

Fig. 5.2. The dots represent integer solutions. The thin dotted line represents a cutting-plane – a constraint that does not remove any integral solution

remain satisfying, as demonstrated in Fig. 5.2. These new constraints improve the tightness of the relaxation in the process of solving integer linear systems.

Here, we describe a family of cutting planes called **Gomory cuts**. We first illustrate this technique with an example, and then generalize it.

Suppose that our problem includes the integer variables $x_1, \ldots, x_3$, and the lower bounds $1 \leq x_1$ and $0.5 \leq x_2$. Further, suppose that the final tableau of the general simplex algorithm includes the constraint

$$x_3 = 0.5x_1 + 2.5x_2 \ , \tag{5.22}$$

and that the solution α is $\{x_3 \mapsto 1.75, x_1 \mapsto 1, x_2 \mapsto 0.5\}$, which, of course, satisfies (5.22). Subtracting these values from (5.22) gives us

$$x_3 - 1.75 = 0.5(x_1 - 1) + 2.5(x_2 - 0.5) \ . \tag{5.23}$$

We now wish to rewrite this equation so the left-hand side is an integer:

$$x_3 - 1 = 0.75 + 0.5(x_1 - 1) + 2.5(x_2 - 0.5) \ . \tag{5.24}$$

The two right-most terms must be positive because 1 and 0.5 are the lower bounds of x_1 and x_2, respectively. Since the right-hand side must add up to an integer as well, this implies that

$$0.75 + 0.5(x_1 - 1) + 2.5(x_2 - 0.5) \geq 1 \ . \tag{5.25}$$

Note, however, that this constraint is unsatisfied by α since by construction all the elements on the left other than the fraction 0.75 are equal to zero under α. This means that adding this constraint to the relaxed system will rule out this solution. On the other hand since it is implied by the integer system of constraints, it cannot remove any *integer* solution.

Let us generalize this example into a recipe for generating such cutting planes. The generalization refers also to the case of having variables assigned their upper bounds, and both negative and positive coefficients. In order to derive a Gomory cut from a constraint, the constraint has to satisfy two conditions: First, the assignment to the basic variable has to be fractional; Second, the assignments to all the nonbasic variables have to correspond to one of their bounds. The following recipe, which relies on these conditions, is based on a report by Dutertre and de Moura [71].

Consider the i-th constraint:

$$x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \ , \tag{5.26}$$

where $x_i \in \mathcal{B}$. Let α be the assignment returned by the general simplex algorithm. Thus,

$$\alpha(x_i) = \sum_{x_j \in \mathcal{N}} a_{ij} \alpha(x_j) \ . \tag{5.27}$$

We now partition the nonbasic variables to those that are currently assigned their lower bound and those that are currently assigned their upper bound:

$$\begin{aligned} J &= \{j \mid x_j \in \mathcal{N} \wedge \alpha(x_j) = l_j\} \\ K &= \{j \mid x_j \in \mathcal{N} \wedge \alpha(x_j) = u_j\} \ . \end{aligned} \tag{5.28}$$

Subtracting (5.27) from (5.26) taking the partition into account yields

$$x_i - \alpha(x_i) = \sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) \ . \tag{5.29}$$

Let $f_0 = \alpha(x_i) - \lfloor \alpha(x_i) \rfloor$. Since we assumed that $\alpha(x_i)$ is not an integer then $0 < f_0 < 1$. We can now rewrite (5.29) as

$$x_i - \lfloor \alpha(x_i) \rfloor = f_0 + \sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) \ . \tag{5.30}$$

Note that the left-hand side is an integer. We now consider two cases.

- If $\sum_{j\in J} a_{ij}(x_j - l_j) - \sum_{j\in K} a_{ij}(u_j - x_j) > 0$ then, since the right-hand side must be an integer,

$$f_0 + \sum_{j\in J} a_{ij}(x_j - l_j) - \sum_{j\in K} a_{ij}(u_j - x_j) \geq 1 \,. \tag{5.31}$$

We now split J and K as follows:

$$\begin{aligned} J^+ &= \{j \mid j \in J \wedge a_{ij} > 0\} \\ J^- &= \{j \mid j \in J \wedge a_{ij} < 0\} \\ K^+ &= \{j \mid j \in K \wedge a_{ij} > 0\} \\ J^- &= \{j \mid j \in K \wedge a_{ij} < 0\} \end{aligned} \tag{5.32}$$

Gathering only the positive elements in the left-hand side of (5.31) gives us:

$$\sum_{j\in J^+} a_{ij}(x_j - l_j) - \sum_{j\in K^-} a_{ij}(u_j - x_j) \geq 1 - f_0 \,, \tag{5.33}$$

or, equivalently,

$$\sum_{j\in J^+} \frac{a_{ij}}{1-f_0}(x_j - l_j) - \sum_{j\in K^-} \frac{a_{ij}}{1-f_0}(u_j - x_j) \geq 1 \,. \tag{5.34}$$

- If $\sum_{j\in J} a_{ij}(x_j - l_j) - \sum_{j\in K} a_{ij}(u_j - x_j) \leq 0$ then again, since the right-hand side must be an integer,

$$f_0 + \sum_{j\in J} a_{ij}(x_j - l_j) - \sum_{j\in K} a_{ij}(u_j - x_j) \leq 0 \,. \tag{5.35}$$

Eq. (5.35) implies that

$$\sum_{j\in J^-} a_{ij}(x_j - l_j) - \sum_{j\in K^+} a_{ij}(u_j - x_j) \leq -f_0 \,. \tag{5.36}$$

Dividing by $-f_0$ gives us

$$-\sum_{j\in J^-} \frac{a_{ij}}{f_0}(x_j - l_j) + \sum_{j\in K^+} \frac{a_{ij}}{f_0}(u_j - x_j) \geq 1 \,. \tag{5.37}$$

Note that the left-hand side of both (5.34) and (5.37) is greater than zero. Therefore these two equations imply

$$\begin{aligned} &\sum_{j\in J^+} \frac{a_{ij}}{1-f_0}(x_j - l_j) - \sum_{j\in J^-} \frac{a_{ij}}{f_0}(x_j - l_j) \\ &+ \sum_{j\in K^+} \frac{a_{ij}}{f_0}(u_j - x_j) - \sum_{j\in K^-} \frac{a_{ij}}{1-f_0}(u_j - x_j) \geq 1 \,. \end{aligned} \tag{5.38}$$

Since each of the elements on the left-hand side is equal to zero under the current assignment α, this assignment α is ruled out by the new constraint. In other words, the solution to the linear problem augmented with the constraint is guaranteed to be different from the previous one.

5.4 Fourier–Motzkin Variable Elimination

5.4.1 Equality Constraints

Similarly to the simplex method, the Fourier–Motzkin variable elimination algorithm takes a conjunction of linear constraints over real variables. Let m denote the number of such constraints, and let $x_1, \ldots, x_n$ denote the variables used by these constraints.

As a first step, equality constraints of the following form are eliminated:

$$\sum_{j=1}^{n} a_{i,j} \cdot x_j = b_i \ . \tag{5.39}$$

We choose a variable x_j that has a nonzero coefficient $a_{i,j}$ in an equality constraint i. Without loss of generality, we assume that x_n is the variable that is to be eliminated. The constraint (5.39) can be rewritten as

$$x_n \ = \ \frac{b_i}{a_{i,n}} - \sum_{j=1}^{n-1} \frac{a_{i,j}}{a_{i,n}} \cdot x_j \ . \tag{5.40}$$

Now we substitute the right-hand side of (5.40) for x_n into all the other constraints, and remove constraint i. This is iterated until all equalities are removed.

We are left with a system of inequalities of the form

$$\bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{i,j} x_j \leq b_i \ . \tag{5.41}$$

5.4.2 Variable Elimination

The basic idea of the variable elimination algorithm is to heuristically choose a variable and then to eliminate it by projecting its constraints onto the rest of the system, resulting in new constraints.

Example 5.9. Consider Fig. 5.3(a): the constraints

$$0 \leq x \leq 1, \ 0 \leq y \leq 1, \ \frac{3}{4} \leq z \leq 1 \tag{5.42}$$

form a cuboid. Projecting these constraints onto the x and y axes, and thereby eliminating z, results in a square which is given by the constraints

$$0 \leq x \leq 1, \ 0 \leq y \leq 1 \ . \tag{5.43}$$

Figure 5.3(b) shows a triangle formed by the constraints

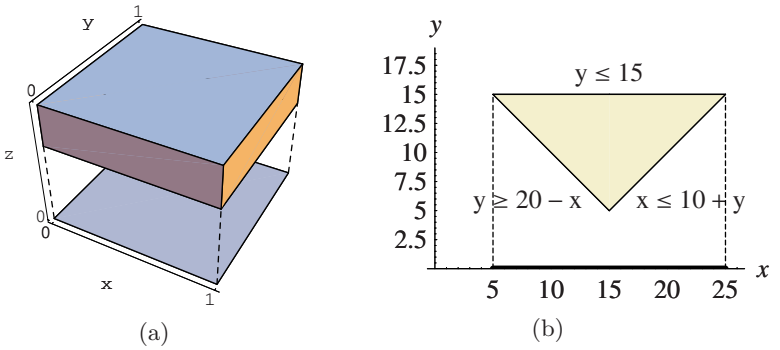

Fig. 5.3. Projection of constraints: (a) a cuboid is projected onto the x and y axes; (b) a triangle is projected onto the x axis

$$x \leq y + 10,\ y \leq 15,\ y \geq -x + 20\ . \tag{5.44}$$

The projection of the triangle onto the x axis is a line given by the constraints

$$5 \leq x \leq 25\ . \tag{5.45}$$

Thus, the projection forms a new problem with one variable fewer, but possibly more constraints. This is done iteratively until all variables but one have been eliminated. The problem with one variable is trivially decidable.

The order in which the variables are eliminated may be predetermined, or adjusted dynamically to the current set of constraints. There are various heuristics for choosing the elimination order. A standard greedy heuristic gives priority to variables that produce fewer new constraints when eliminated.

Once again, assume that x_n is the variable chosen to be eliminated. The constraints are partitioned according to the coefficient of x_n. Consider the constraint with index i:

$$\sum_{j=1}^{n} a_{i,j} \cdot x_j \leq b_i\ . \tag{5.46}$$

By splitting the sum, (5.46) can be rewritten into

$$a_{i,n} \cdot x_n \ \leq\ b_i - \sum_{j=1}^{n-1} a_{i,j} \cdot x_j\ . \tag{5.47}$$

If $a_{i,n}$ is zero, the constraint can be disregarded when we are eliminating x_n. Otherwise, we divide by $a_{i,n}$. If $a_{i,n}$ is positive, we obtain

$$x_n \ \leq\ \frac{b_i}{a_{i,n}} - \sum_{j=1}^{n-1} \frac{a_{i,j}}{a_{i,n}} \cdot x_j\ . \tag{5.48}$$

β_i

Thus, if $a_{i,n} > 0$, the constraint is an *upper bound* on x_n. If $a_{i,n} < 0$, the constraint is a *lower bound*. We denote the right-hand side of (5.48) by β_i.

Unbounded Variables

It is possible that a variable is not bounded both ways, i.e., it has either only upper bounds or only lower bounds. Such variables are called **unbounded variables**. Unbounded variables can be simply removed from the system together with all constraints that use them. Removing these constraints can make other variables unbounded. Thus, this simplification stage iterates until no such variables remain.

Bounded Variables

If x_n has both an upper and a lower bound, the algorithm enumerates all pairs of lower and upper bounds. Let $u \in \{1, \ldots, m\}$ denote the index of an upper-bound constraint, and $l \in \{1, \ldots, m\}$ denote the index of a lower-bound constraint for x_n, where $l \neq u$. For each such pair, we have

$$\beta_l \quad \leq \quad x_n \quad \leq \quad \beta_u \; . \tag{5.49}$$

The following new constraint is added:

$$\beta_l \quad \leq \quad \beta_u \; . \tag{5.50}$$

The Formula (5.50) may simplify to $0 \leq b_k$, where b_k is some constant smaller than 0. In this case, the algorithm has found a *conflicting* pair of constraints and concludes that the problem is unsatisfiable. Otherwise, all constraints that involve x_n are removed. The new problem is solved recursively as before.

Example 5.10. Consider the following set of constraints:

$$\begin{array}{rrrrr} x_1 & -x_2 & & \leq & 0 \\ x_1 & & -x_3 & \leq & 0 \\ -x_1 & +x_2 & +2x_3 & \leq & 0 \\ & & -x_3 & \leq & -1 \; . \end{array} \tag{5.51}$$

Suppose we decide to eliminate the variable x_1 first. There are two upper bounds on x_1, namely $x_1 \leq x_2$ and $x_1 \leq x_3$, and one lower bound, which is $x_2 + 2x_3 \leq x_1$.

Using $x_1 \leq x_2$ as the upper bound, we obtain a new constraint $2x_3 \leq 0$, and using $x_1 \leq x_3$ as the upper bound, we obtain a new constraint $x_2 + x_3 \leq 0$. Constraints involving x_1 are removed from the problem, which results in the following new set:

$$\begin{array}{rrrr} 2x_3 & \leq & 0 \\ x_2 + x_3 & \leq & 0 \\ -x_3 & \leq & -1 \; . \end{array} \tag{5.52}$$

Next, observe that x_2 is unbounded (as it has no lower bound), and hence the second constraint can be eliminated, which simplifies the formula. We therefore progress by eliminating x_2 and all the constraints that contain it:

$$\begin{array}{rl} 2x_3 \leq & 0 \\ -x_3 \leq & -1 \ . \end{array} \tag{5.53}$$

Only the variable x_3 remains, with a lower and an upper bound. Combining the two into a new constraint results in $1 \leq 0$, which is a contradiction. Thus, the system is unsatisfiable. ⌟

The simplex method in its basic form, as described in Sect. 5.2, allows only nonstrict ($\leq$) inequalities.[4] The Fourier–Motzkin method, on the other hand, can easily be extended to handle a combination of strict ($<$) and nonstrict inequalities: if either the lower or the upper bound is a strict inequality, then so is the resulting constraint.

5.4.3 Complexity

In each iteration, the number of constraints can increase in the worst case from m to $m^2/4$, which results overall in $m^{2^n}/4^n$ constraints. Thus, Fourier–Motzkin variable elimination is only suitable for a relatively small set of constraints and a small number of variables.

5.5 The Omega Test

5.5.1 Problem Description

The Omega test is an algorithm to decide the satisfiability of a conjunction of linear constraints over integer variables. Each conjunct is assumed to be either an equality of the form

$$\sum_{i=1}^{n} a_i x_i = b \tag{5.54}$$

or a nonstrict inequality of the form

$$\sum_{i=1}^{n} a_i x_i \leq b \ . \tag{5.55}$$

The coefficients a_i are assumed to be integers; if they are not, by making use of the assumption that the coefficients are rational, the problem can be transformed into one with integer coefficients by multiplying the constraints

[4] There are extensions of Simplex to strict inequalities. See, for example, [70].

by the least common multiple of the denominators. In Sect. 5.6, we show how strict inequalities can be transformed into nonstrict inequalities.

The runtime of the Omega test depends on the size of the coefficients a_i. It is therefore desirable to transform the constraints such that small coefficients are obtained. This can be done by dividing the coefficients $a_1, \ldots, a_n$ of each constraint by their greatest common divisor g. The resulting constraint is called *normalized*. If the constraint is an equality constraint, this results in

$$\sum_{i=1}^{n} \frac{a_i}{g} x_i = \frac{b}{g} \,. \tag{5.56}$$

If g does not divide b exactly, the system is unsatisfiable. If the constraint is an inequality, one can tighten the constraint by rounding down the constant:

$$\sum_{i=1}^{n} \frac{a_i}{g} x_i \leq \left\lfloor \frac{b}{g} \right\rfloor . \tag{5.57}$$

More simplifications of this kind are described in Sect. 5.6.

Example 5.11. The equality $3x + 3y = 2$ can be normalized to $x + y = 2/3$, which is unsatisfiable. The constraint $8x + 6y \leq 0$ can be normalized to obtain $4x + 3y \leq 0$. The constraint $1 \leq 4y$ can be tightened to obtain $1 \leq y$. ⌟

The Omega test is a variant of the Fourier–Motzkin variable elimination algorithm (Sect. 5.4). As in the case of that algorithm, equality and inequality constraints are treated separately; all equality constraints are removed before inequalities are considered.

5.5.2 Equality Constraints

In order to eliminate an equality of the form of (5.54), we first check if there is a variable x_j with a coefficient 1 or -1, i.e., $|a_j| = 1$. If yes, we transform the constraint as follows. Without loss of generality, assume $j = n$. We isolate x_n:

$$x_n = \frac{b}{a_n} - \sum_{i=1}^{n-1} \frac{a_i}{a_n} x_i \,. \tag{5.58}$$

The variable x_n can now be substituted by the right-hand side of (5.58) in all constraints.

If there is no variable with a coefficient 1 or -1, we cannot simply divide by the coefficient, as this would result in nonintegral coefficients. Instead, the algorithm proceeds as follows: it determines the variable that has the nonzero coefficient with the smallest absolute value. Assume again that x_n is chosen, and that $a_n > 0$. The Omega test transforms the constraints iteratively until some coefficient becomes 1 or -1. The variable with that coefficient can then be eliminated as above.

$a \mathbin{\widehat{\bmod}} b$

For this transformation, a new binary operator $\mathbin{\widehat{\bmod}}$, called **symmetric modulo**, is defined as follows:

$$a \mathbin{\widehat{\bmod}} b \doteq a - b \cdot \left\lfloor \frac{a}{b} + \frac{1}{2} \right\rfloor . \tag{5.59}$$

The symmetric modulo operator is very similar to the usual modular arithmetic operator. If $a \bmod b < b/2$, then $a \mathbin{\widehat{\bmod}} b = a \bmod b$. If $a \bmod b$ is greater than or equal to $b/2$, b is deducted, and thus

$$a \mathbin{\widehat{\bmod}} b = \begin{cases} a \bmod b & : a \bmod b < b/2 \\ (a \bmod b) - b & : \text{otherwise} . \end{cases} \tag{5.60}$$

We leave the proof of this equivalence as an exercise (see Problem 5.12).

Our goal is to derive a term that can replace x_n. For this purpose, we define $m \doteq a_n + 1$, introduce a new variable σ, and add the following new constraint:

$$\sum_{i=1}^{n} (a_i \mathbin{\widehat{\bmod}} m)\, x_i = m\sigma + b \mathbin{\widehat{\bmod}} m . \tag{5.61}$$

We split the sum on the left-hand side to obtain

$$(a_n \mathbin{\widehat{\bmod}} m) x_n = m\sigma + b \mathbin{\widehat{\bmod}} m - \sum_{i=1}^{n-1} (a_i \mathbin{\widehat{\bmod}} m)\, x_i . \tag{5.62}$$

Since $a_n \mathbin{\widehat{\bmod}} m = -1$ (see Problem 5.14), this simplifies to:

$$x_n = -m\sigma - b \mathbin{\widehat{\bmod}} m + \sum_{i=1}^{n-1} (a_i \mathbin{\widehat{\bmod}} m)\, x_i . \tag{5.63}$$

The right-hand side of (5.63) is used to replace x_n in all constraints. Any equality from the original problem (5.54) is changed as follows:

$$\sum_{i=1}^{n-1} a_i x_i + a_n \left(-m\sigma - b \mathbin{\widehat{\bmod}} m + \sum_{i=1}^{n-1} (a_i \mathbin{\widehat{\bmod}} m)\, x_i \right) = b , \tag{5.64}$$

which can be rewritten as

$$-a_n m\sigma + \sum_{i=1}^{n-1} (a_i + a_n (a_i \mathbin{\widehat{\bmod}} m))\, x_i = b + a_n (b \mathbin{\widehat{\bmod}} m) . \tag{5.65}$$

Since $a_n = m - 1$, this simplifies to

$$\begin{aligned} &-a_n m\sigma + \textstyle\sum_{i=1}^{n-1} ((a_i - (a_i \mathbin{\widehat{\bmod}} m)) + m(a_i \mathbin{\widehat{\bmod}} m))\, x_i = \\ &b - (b \mathbin{\widehat{\bmod}} m) + m(b \mathbin{\widehat{\bmod}} m) . \end{aligned} \tag{5.66}$$

Note that $a_i - (a_i \widehat{\bmod} m)$ is equal to $m\lfloor a_i/m + 1/2 \rfloor$, and thus all terms are divisible by m. Dividing (5.66) by m results in

$$-a_n\sigma + \sum_{i=1}^{n-1} (\lfloor a_i/m + 1/2 \rfloor + (a_i \widehat{\bmod} m))\, x_i = \lfloor b/m + 1/2 \rfloor + (b \widehat{\bmod} m) \ . \quad (5.67)$$

The absolute value of the coefficient of σ is the same as the absolute value of the original coefficient a_n, and it seems that nothing has been gained by this substitution. However, observe that the coefficient of x_i can be bounded as follows (see Problem 5.13):

$$|\lfloor a_i/m + 1/2 \rfloor + (a_i \widehat{\bmod} m)| \leq \frac{5}{6}|a_i| \ . \quad (5.68)$$

Thus, the absolute values of the coefficients in the equality are strictly smaller than their previous values. As the coefficients are always integral, repeated application of equality elimination eventually generates a coefficient of 1 or -1 on some variable. This variable can then be eliminated directly, as described earlier (see (5.58)).

Example 5.12. Consider the following formula:

$$\begin{aligned} -3x_1 + 2x_2 &= 0 \\ 3x_1 + 4x_2 &= 3 \ . \end{aligned} \quad (5.69)$$

The variable x_2 has the coefficient with the smallest absolute value ($a_2 = 2$). Thus, $m = a_2 + 1 = 3$, and we add the following constraint (see (5.61)):

$$(-3 \widehat{\bmod} 3)x_1 + (2 \widehat{\bmod} 3)x_2 = 3\sigma \ . \quad (5.70)$$

This simplifies to $x_2 = -3\sigma$. Substituting -3σ for x_2 results in the following problem:

$$\begin{aligned} -3x_1 - 6\sigma &= 0 \\ 3x_1 - 12\sigma &= 3 \ . \end{aligned} \quad (5.71)$$

Division by m results in

$$\begin{aligned} -x_1 - 2\sigma &= 0 \\ x_1 - 4\sigma &= 1 \ . \end{aligned} \quad (5.72)$$

As expected, the coefficient of x_1 has decreased. We can now substitute x_1 by $4\sigma + 1$, and obtain $-6\sigma = 1$, which is unsatisfiable. ⌟

5.5.3 Inequality Constraints

Once all equalities have been eliminated, the algorithm attempts to find a solution for the remaining inequalities. The control flow of Algorithm 5.5.1 is illustrated in Fig. 5.4. As in the Fourier–Motzkin procedure, the first step is to choose a variable to be eliminated. Subsequently, the three subprocedures

Real-Shadow, *Dark-Shadow*, and *Gray-Shadow* produce new constraint sets, which are solved recursively.

Note that many of the subproblems generated by the recursion are actually identical. An efficient implementation uses a hash table that stores the solutions of previously solved problems.

Algorithm 5.5.1: OMEGA-TEST

Input: A conjunction of constraints C
Output: "Satisfiable" if C is satisfiable, and "Unsatisfiable" otherwise

```
 1. if C only contains one variable then
 2.     Solve and return result;              ▷ (solving this problem is trivial)
 3.
 4. Otherwise, choose a variable v that occurs in C;
 5. C_R := Real-Shadow(C, v);
 6. if Omega-Test(C_R) = "Unsatisfiable" then          ▷ Recursive call
 7.     return "Unsatisfiable";
 8.
 9. C_D := Dark-Shadow(C, v);
10. if Omega-Test(C_D) = "Satisfiable" then            ▷ Recursive call
11.     return "Satisfiable";
12.
13. if C_R = C_D then                                  ▷ Exact projection?
14.     return "Unsatisfiable";
15.
16. C_G^1, ..., C_G^n := Gray-Shadow(C, v);
17. for all i ∈ {1, ..., n} do
18.     if Omega-Test(C_G^i) = "Satisfiable" then      ▷ Recursive call
19.         return "Satisfiable";
20.
21. return "Unsatisfiable";
```

Checking the Real Shadow

Even though the Omega test is concerned with constraints over integers, the first step is to check if there are integer solutions in the relaxed problem, which is called the *real shadow*. The real shadow is the same projection that the Fourier–Motzkin procedure uses. The Omega test is then called recursively to check if the projection contains an integer. If there is no such integer, then there is no integer solution to the original system either, and the algorithm concludes that the system is unsatisfiable.

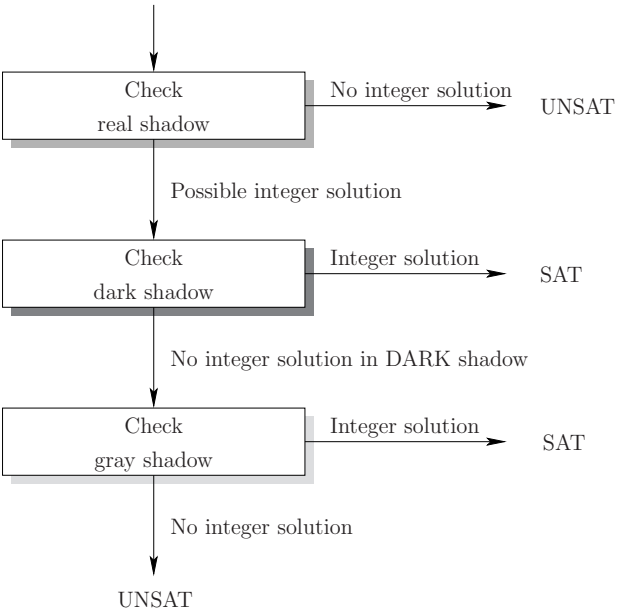

Fig. 5.4. Overview of the Omega test

Assume that the variable to be eliminated is denoted by z. As in the case of the Fourier–Motzkin procedure, all pairs of lower and upper bounds have to be considered. Variables that are not bounded both ways can be removed, together with all constraints that contain them.

Let $\beta \leq bz$ and $cz \leq \gamma$ be constraints, where c and b are positive integer constants and γ and β denote the remaining linear expressions. Consequently, β/b is a lower bound on z, and γ/c is an upper bound on z. The new constraint is obtained by multiplying the lower bound by c and the upper bound by b:

$$\begin{array}{ll} \text{Lower bound} & \text{Upper bound} \\ \hline \beta \leq bz & cz \leq \gamma \\ c\beta \leq cbz & cbz \leq b\gamma \end{array} \tag{5.73}$$

The existence of such a variable z implies

$$c\beta \leq b\gamma \; . \tag{5.74}$$

Example 5.13. Consider the following set of constraints:

$$\begin{array}{rl} 2y \leq & x \\ 8y \geq 2 + & x \\ 2y \leq 3 - & x \; . \end{array} \tag{5.75}$$

The triangle spanned by these constraints is depicted in Fig. 5.5. Assume that we decide to eliminate x. In this case, the combination of the two constraints

$2y \leq x$ and $8y \geq 2 + x$ results in $8y - 2 \geq 2y$, which simplifies to $y \geq 1/3$. The two constraints $2y \leq x$ and $2y \leq 3 - x$ combine into $2y \leq 3 - 2y$, which simplifies to $y \leq 3/4$. Thus, $1/3 \leq y \leq 3/4$ must hold, which has no integer solution. The set of constraints is therefore unsatisfiable. ⌟

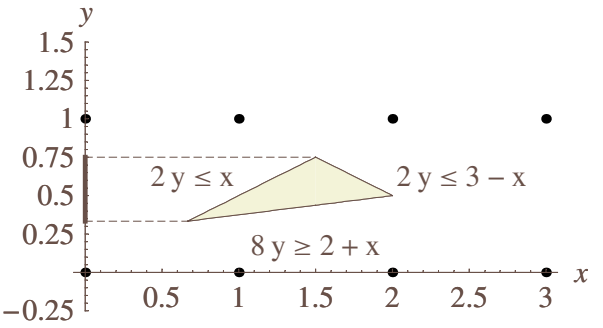

Fig. 5.5. Computing the real shadow: eliminating x

The converse of this observation does not hold, i.e., if we find an integer solution within the real shadow, this does not guarantee that the original set of constraints has an integer solution. This is illustrated by the following example.

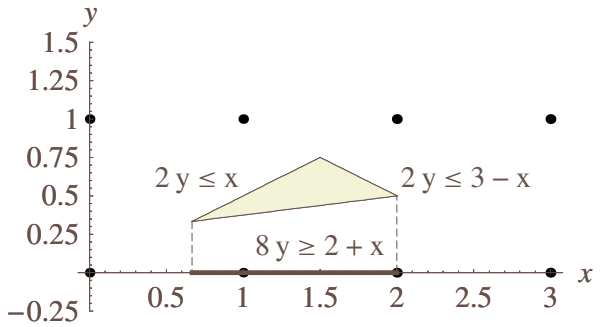

Fig. 5.6. Computing the real shadow: eliminating y

Example 5.14. Consider the same set of constraints as in Example 5.13. This time, eliminate y instead of x. This projection is depicted in Fig. 5.6.

We obtain $2/3 \leq x \leq 2$, which has two integer solutions. The triangle, on the other hand, contains no integer solution. ⌟

The real shadow is an overapproximating projection, as it contains more solutions than does the original problem. The next step in the Omega test is to compute an underapproximating projection, i.e., if that projection contains an integer solution, so does the original problem. This projection is called the *dark* shadow.

Checking the Dark Shadow

The name *dark shadow* is motivated by optics. Assume that the object we are projecting is partially translucent. Places that are "thicker" will project a darker shadow. In particular, a dark area in the shadow where the object is thicker than 1 must have at least one integer above it.

After the first phase of the algorithm, we know that there is a solution to the real shadow, i.e., $c\beta \leq b\gamma$. We now aim at determining if there is an integer z such that $c\beta \leq cbz \leq b\gamma$, which is equivalent to

$$\exists z \in \mathbb{Z}.\ \frac{\beta}{b} \leq z \leq \frac{\gamma}{c}\ . \tag{5.76}$$

Assume that (5.76) does not hold. Let i denote $\lfloor \beta/b \rfloor$, i.e., the largest integer that is smaller than β/b. Since we have assumed that there is no integer between β/b and γ/c,

$$i < \frac{\beta}{b} \leq \frac{\gamma}{c} < i + 1 \tag{5.77}$$

holds. This situation is illustrated in Fig. 5.7.

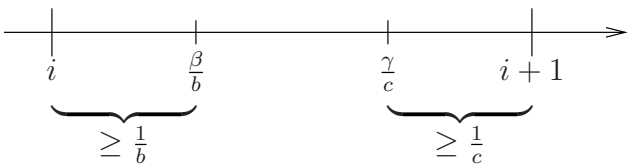

Fig. 5.7. Computing the dark shadow

Since β/b and γ/c are not integers themselves, the distances from these points to the closest integer are greater than the fractions $1/b$ and $1/c$, respectively:

$$\frac{\beta}{b} - i \geq \frac{1}{b} \tag{5.78}$$

$$i + 1 - \frac{\gamma}{c} \geq \frac{1}{c}\ . \tag{5.79}$$

The proof is left as an exercise (Problem 5.11). By summing (5.78) and (5.79), we obtain

$$\frac{\beta}{b} + 1 - \frac{\gamma}{c} \geq \frac{1}{c} + \frac{1}{b} \,, \tag{5.80}$$

which is equivalent to

$$c\beta - b\gamma \geq -cb + c + b \,. \tag{5.81}$$

By multiplying this inequality by -1, we obtain

$$b\gamma - c\beta \leq cb - c - b \,. \tag{5.82}$$

In order to show a contradiction to our assumption, we need to show the negation of (5.82). Exploiting the fact that c, b are integers, the negation of (5.82) is

$$b\gamma - c\beta \geq cb - c - b + 1 \,, \tag{5.83}$$

or simply

$$b\gamma - c\beta \geq (c-1)(b-1) \,. \tag{5.84}$$

Thus, if (5.84) holds, our assumption is wrong, which means that we have a guarantee that there exists an integer solution.

Observe that if either $c = 1$ or $b = 1$, the formula (5.84) is identical to the real shadow (5.74), i.e., the dark and real shadow are the same. In this case, the projection is exact, and it is sufficient to check the *real shadow*. When choosing variables to eliminate, preference should be given to variables that result in an exact projection, that is, to variables with coefficient 1.

Checking the Gray Shadow

We know that any integer solution must also be in the real shadow. Let **R** denote this area. Now assume that we have found no integer in the dark shadow. Let **D** denote the area of the dark shadow.

R

D

Thus, if **R** and **D** do not coincide, there is only one remaining area in which an integer solution can be found: an area around the dark shadow, which, staying within the optical analogy, is called the *gray shadow*.

Any solution must satisfy

$$c\beta \leq cbz \leq b\gamma \,. \tag{5.85}$$

Furthermore, we already know that the dark shadow does not contain an integer, and thus we can exclude this area from the search. Therefore, besides (5.85), any solution has to satisfy (5.82):

$$c\beta \leq cbz \leq b\gamma \quad \wedge \quad b\gamma - c\beta \leq cb - c - b \; . \tag{5.86}$$

This is equivalent to

$$c\beta \leq cbz \leq b\gamma \quad \wedge \quad b\gamma \leq cb - c - b + c\beta \; , \tag{5.87}$$

which implies

$$c\beta \leq cbz \leq cb - c - b + c\beta \; . \tag{5.88}$$

Dividing by c, we obtain

$$\beta \leq bz \leq \beta + \frac{cb - c - b}{c} \; . \tag{5.89}$$

The Omega test proceeds by simply trying possible values of bz between these two bounds. Thus, a new constraint

$$bz = \beta + i \tag{5.90}$$

is formed and combined with the original problem for each integer i in the range $0, \ldots, (cb - c - b)/c$. If any one of the resulting new problems has a solution, so does the original problem.

The number of subproblems can be reduced by determining the largest coefficient c of z in any upper bound for z. The new constraints generated for the other upper bounds are already covered by the constraints generated for the upper bound with the largest c.

5.6 Preprocessing

In this section, we examine several simple preprocessing steps for both linear and integer linear systems without objective functions. Preprocessing the set of constraints can be done regardless of the decision procedure chosen.

5.6.1 Preprocessing of Linear Systems

Two simple preprocessing steps for linear systems are the following:

1. Consider the set of constraints

$$x_1 + x_2 \leq 2, \; x_1 \leq 1, \; x_2 \leq 1 \; . \tag{5.91}$$

The first constraint is redundant. In general, for a set:

$$S = \left\{ a_0 x_0 + \sum_{j=1}^{n} a_j x_j \leq b, \; l_j \leq x_j \leq u_j \text{ for } j = 0, \ldots, n \right\} \; , \tag{5.92}$$

the constraint

$$a_0 x_0 + \sum_{j=1}^{n} a_j x_j \leq b \tag{5.93}$$

is redundant if

$$\sum_{j|a_j>0} a_j u_j + \sum_{j|a_j<0} a_j l_j \leq b\,. \tag{5.94}$$

To put this in words, a "$\leq$" constraint in the above form is redundant if assigning values equal to their upper bounds to all of its variables that have a positive coefficient, and assigning values equal to their lower bounds to all of its variables that have a negative coefficient, results in a value less than or equal to b, the constant on the right-hand side of the inequality.

2. Consider the following set of constraints:

$$2x_1 + x_2 \leq 2,\ x_2 \geq 4,\ x_1 \leq 3\,. \tag{5.95}$$

From the first and second constraints, $x_1 \leq -1$ can be derived, which means that the bound $x_1 \leq 3$ can be tightened. In general, if $a_0 > 0$, then

$$x_0 \leq \left(b - \sum_{j|j>0,a_j>0} a_j l_j - \sum_{j|a_j<0} a_j u_j \right) / a_0\,, \tag{5.96}$$

and if $a_0 < 0$, then

$$x_0 \geq \left(b - \sum_{j|a_j>0} a_j l_j - \sum_{j|j>0,a_j<0} a_j u_j \right) / a_0\,. \tag{5.97}$$

5.6.2 Preprocessing of Integer Linear Systems

The following preprocessing steps are applicable to integer linear systems:

1. Multiply every constraint by the smallest common multiple of the coefficients and constants in this constraint, in order to obtain a system with integer coefficients.[5]
2. After the previous preprocessing has been applied, strict inequalities can be transformed into nonstrict inequalities as follows:

$$\sum_{1\leq i\leq n} a_i x_i < b \tag{5.98}$$

is replaced with

$$\sum_{1\leq i\leq n} a_i x_i \leq b - 1\,. \tag{5.99}$$

The case in which b is fractional is handled by the previous preprocessing step.

[5] This assumes that the coefficients and constants in the system are rational. The case in which the coefficients can be nonrational is of little value and is rarely considered in the literature.

For the special case of **0–1 linear systems** (integer linear systems in which all the variables are constrained to be either 0 or 1), some preprocessing steps are illustrated by the following examples:

1. Consider the constraint
$$5x_1 - 3x_2 \leq 4\,, \tag{5.100}$$
from which we can conclude that
$$x_1 = 1 \implies x_2 = 1\,. \tag{5.101}$$
Hence, the constraint
$$x_1 \leq x_2 \tag{5.102}$$
can be added.
2. From
$$x_1 + x_2 \leq 1,\ x_2 \geq 1\,, \tag{5.103}$$
we can conclude $x_1 = 0$.

Generalization of these examples is left for Problem 5.8.

5.7 Difference Logic

5.7.1 Introduction

A popular fragment of linear arithmetic is called **difference logic**.

Definition 5.15 (difference logic). *The syntax of a formula in* difference logic *is defined by the following rules:*

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \mathit{atom} \\ \mathit{atom} &: \mathit{identifier} - \mathit{identifier}\ \mathit{op}\ \mathit{constant} \\ \mathit{op} &: \leq \mid < \end{aligned}$$

Here, we consider the case in which the variables are defined over $\mathbb{Q}$, the rationals. A similar definition exists for the case in which the variables are defined over $\mathbb{Z}$ (see Problem 5.18). Solving both variants is polynomial, whereas, recall, linear arithmetic over $\mathbb{Z}$ is NP-complete.

Some other convenient operands can be modeled with the grammar above:

- $x - y = c$ is the same as $x - y \leq c \wedge y - x \leq -c$.
- $x - y \geq c$ is the same as $y - x \leq -c$.
- $x - y > c$ is the same as $y - x < -c$.
- A constraint with one variable such as $x < 5$ can be rewritten as $x - x_0 < 5$, where x_0 is a special variable not used so far in the formula, called the "zero variable". In any satisfying assignment, its value must be 0.

As an example,

$$x < y + 5 \wedge y \leq 4 \wedge x = z - 1 \tag{5.104}$$

can be rewritten in difference logic as

$$x - y < 5 \wedge y - x_0 \leq 4 \wedge x - z \leq -1 \wedge z - x \leq 1. \tag{5.105}$$

A more important variant, however, is one in which an arbitrary Boolean structure is permitted. We describe one application of this variant by the following example.

Example 5.16. We are given a finite set of n jobs, each of which consists of a chain of operations. There is a finite set of m machines, each of which can handle at most one operation at a time. Each operation needs to be performed during an uninterrupted period of given length on a given machine. The **job-shop scheduling** problem is to find a schedule, that is, an allocation of the operations to time intervals on the machines that has a minimal total length.

More formally, given a set of machines

$$M = \{m_1, \ldots, m_m\} , \tag{5.106}$$

job J^i with $i \in \{1, \ldots, n\}$ is a sequence of n_i pairs of the form (machine, duration):

$$J^i = (m_1^i, d_1^i), \ldots, (m_{n_i}^i, d_{n_i}^i) , \tag{5.107}$$

such that $m_1^i, \ldots, m_{n_i}^i$ are elements of M. The durations can be assumed to be rational numbers. We denote by O the multiset of all operations from all jobs. For an operation $v \in O$, we denote its machine by $M(v)$ and its duration by $\tau(v)$.

A schedule is a function that defines, for each operation v, its starting time $S(v)$ on its specified machine $M(v)$. A schedule S is *feasible* if the following three constraints hold.

First, the starting time of all operations is greater than or equal to 0:

$$\forall v \in O.\ S(v) \geq 0 . \tag{5.108}$$

Second, for every pair of consecutive operations $v_i, v_j \in O$ in the same job, the second operation does not start before the first ends:

$$S(v_i) + \tau(v_i) \leq S(v_j) . \tag{5.109}$$

Finally, every pair of different operations $v_i, v_j \in O$ scheduled on the same machine ($M(v_i) = M(v_j)$) is mutually exclusive:

$$S(v_i) + \tau(v_i) \leq S(v_j) \vee S(v_j) + \tau(v_j) \leq S(v_i) . \tag{5.110}$$

The length of the schedule S is defined as

$$\max_{v \in O} S(v) + \tau(v) , \tag{5.111}$$

and the objective is to find a schedule S that minimizes this length. As usual, we can define the decision problem associated with this optimization problem by removing the objective function and adding a constraint that forces the value of this function to be smaller than some constant.

It should be clear that a job-shop scheduling problem can be formulated with difference logic. Note the disjunction in (5.110). ⌟

5.7.2 A Decision Procedure for Difference Logic

Recall that in this chapter we present only decision procedures for conjunctive fragments, and postpone the problem of solving the general case to Chap. 11.

Definition 5.17 (inequality graph for nonstrict inequalities). *Let S be a set of difference predicates and let the* inequality graph $G(V, E)$ *be the graph comprising of one edge (x, y) with weight c for every constraint of the form $x - y \leq c$ in S.*

Given a difference logic formula φ with nonstrict inequalities only, the inequality graph corresponding to the set of difference predicates in φ can be used for deciding φ, on the basis of the following theorem.

Theorem 5.18. *Let φ be a conjunction of difference constraints, and let G be the corresponding inequality graph. Then φ is satisfiable if and only if there is no negative cycle in G.*

The proof of this theorem is left as an exercise (Problem 5.15). The extension of Definition 5.17 and Theorem 5.18 to general difference logic (which includes both strict and nonstrict inequalities) is left as an exercise as well (see Problem 5.16).

By Theorem 5.18, deciding a difference logic formula amounts to searching for a negative cycle in a graph. This can be done with the **Bellman–Ford algorithm** [54] for finding the single-source shortest paths in a directed weighted graph, in time $O(|V| \cdot |E|)$ (to make the graph single-source, we introduce a new node and add an edge with weight 0 from this node to each of the roots of the original graph). Although finding the shortest paths is not our goal, we exploit a side-effect of this algorithm: if there exists a negative cycle in the graph, the algorithm finds it and aborts.

5.8 Problems

5.8.1 Warm-up Exercises

Problem 5.1 (linear systems). Consider the following linear system, which we denote by S:

$$\begin{array}{rcl} x_1 & \geq & -x_2 + \frac{11}{5} \\ x_1 & \leq & x_2 + \frac{1}{2} \\ x_1 & \geq & 3x_2 - 3 \ . \end{array} \tag{5.112}$$

(a) Check with simplex whether S is satisfiable, as described in Sect. 5.2.
(b) Using the Fourier–Motzkin procedure, compute the range within which x_2 has to lie in a satisfying assignment.
(c) Consider a problem S', similar to S, but where the variables are forced to be integer. Check with Branch and Bound whether S' is satisfiable. To solve the relaxed problem, you can use a simplex implementation (there are many of these on the Web).

5.8.2 The Simplex Method

Problem 5.2 (simplex). Compute a satisfying assignment for the following problem using the general simplex method:

$$\begin{array}{rl} 2x_1 + 2x_2 + 2x_3 + 2x_4 \leq & 2 \\ 4x_1 + x_2 + x_3 - 4x_4 \leq & -2 \\ x_1 + 2x_2 + 4x_3 + 2x_4 = & 4 \ . \end{array} \tag{5.113}$$

Problem 5.3 (complexity). Give a conjunction of linear constraints over reals with n variables (that is, the size of the instance is parameterized) such that the number of iterations of the general simplex algorithm is exponential in n.

Problem 5.4 (difference logic with simplex). What is the worst-case run time of the general simplex algorithm if applied to a conjunction of difference logic constraints?

Problem 5.5 (strict inequalities with simplex). Extend the general simplex algorithm with strict inequalities.

Problem 5.6 (soundness). Assume that the general simplex algorithm returns "UNSAT". Show a method for deriving a proof of unsatisfiability.

5.8.3 Integer Linear Systems

Problem 5.7 (complexity of ILP-feasibility). Prove that the feasibility problem for integer linear programming is NP-hard.[6]

Problem 5.8 (0–1 ILP). A 0–1 integer linear system is an integer linear system in which all variables are constrained to be either 0 or 1. Show how a 0–1 integer linear system can be translated to a Boolean formula. What is the complexity of the translation?

[6] In fact it is NP-complete, but membership in NP is more difficult to prove. The proof makes use of a small-model-property argument.

Problem 5.9 (simplifications for 0–1 ILP). Generalize the simplification demonstrated in (5.100)–(5.103).

Problem 5.10 (Gomory cuts). Find Gomory cuts corresponding to the following results from the general simplex algorithm:

1. $x_4 = x_1 - 2.5x_2 + 2x_3$ where $\alpha := \{x_4 \mapsto 3.25, x_1 \mapsto 1, x_2 \mapsto -0.5, x_3 \mapsto 0.5\}$, x_2 and x_3 are at their upper bound and x_1 is at its lower bound.
2. $x_4 = -0.5x_1 - 2x_2 - 3.5x_3$ where $\alpha := \{x_4 \mapsto 0.25, x_1 \mapsto 1, x_2 \mapsto 0.5, x_3 \mapsto 0.5\}$, x_1 and x_3 are at their lower bound and x_2 is at its upper bound.

5.8.4 Omega Test

Problem 5.11 (integer fractions). Show that

$$i + 1 - \frac{\gamma}{c} \geq \frac{1}{c} .$$

Problem 5.12 (eliminating equalities). Show that

$$a \mathbin{\widehat{\bmod}} b = \begin{cases} a \bmod b & : a \bmod b < b/2 \\ (a \bmod b) - b & : \text{otherwise} \end{cases} \tag{5.114}$$

holds. Use the fact that

$$a/b = \lfloor a/b \rfloor + \frac{a \bmod b}{b} .$$

Problem 5.13 (eliminating equalities). Show that the absolute values of the coefficients of the variables x_i are reduced to at most 5/6 of their previous values after substituting σ:

$$|\lfloor a_i/m + 1/2 \rfloor + (a_i \mathbin{\widehat{\bmod}} m)| \leq 5/6|a_i| . \tag{5.115}$$

Problem 5.14 (eliminating equalities). The elimination of x_n relies on the fact that the coefficient of x_n in the newly added constraint is -1. Let a_n denote the coefficient of x_n in the original constraint. Let $m = a_n + 1$, and assume that $a_n \geq 2$. Show that $a_n \mathbin{\widehat{\bmod}} m = -1$.

5.8.5 Difference Logic

Problem 5.15 (difference logic). Prove Theorem 5.18.

Problem 5.16 (inequality graphs for difference logic). Extend Definition 5.17 and Theorem 5.18 to general difference logic formulas (i.e., where both strong and weak inequalities are allowed).

Problem 5.17 (difference logic). Give a reduction of difference logic to SAT. What is the complexity of the reduction?

Problem 5.18 (integer difference logic). Show a reduction from the problem of integer difference logic to difference logic.

5.9 Bibliographic Notes

The Fourier–Motzkin variable elimination algorithm is the earliest documented method for solving linear inequalities. It was discovered in 1826 by Fourier, and rediscovered by Motzkin in 1936.

The simplex method was introduced by Danzig in 1947 [55]. There are several variations of and improvements on this method, most notably the *revised simplex method*, which most industrial implementations use. This variant has an apparent advantage on large and sparse LP problems, which seem to characterize LP problems in practice. The variant of the general simplex algorithm that we presented in Sect. 5.2 was proposed by Dutertre and de Moura [70] in the context of DPLL(T), a technique we describe in Chap. 11. Its main advantage is that it works efficiently with incremental operations, i.e., constraints can be added and removed with little effort.

Linear programs are a very popular modeling formalism for solving a wide range of problems in science and engineering, finance, logistics and so on. See, for example, how LP is used for computing an optimal placement of gates in an integrated circuit [100]. The popularity of this method led to a large industry of LP solvers, some of which are sold for tens of thousands of dollars per copy. A classical reference to linear and integer linear programming is the book by Schrijver [174]. Other resources on the subject that we found useful include publications by Wolsey [204], Hillier and Lieberman [92], and Vanderbei [196].

Gomory cutting-planes are due to a paper published by Ralph Gomory in 1963 [89]. For many years, the operations research community considered Gomory cuts impractical for large problems. There were several refinements of the original method and empirical studies that revived this technique, especially in the context of the related optimization problem. See, for example, the work of Balas et al. [72]. The variant we described is suitable for working with the general simplex algorithm and its description here is based on [71].

The Omega test was introduced by Pugh as a method for deciding integer linear arithmetic within an optimizing compiler [160]. It is an extension of the

Fourier–Motzkin variable elimination. For an example of an application of the Omega test inside a Fortran compiler, see [2]. A much earlier work following similar lines to those of the omega test is by Paul Williams [199]. Williams' work, in turn, is inspired by Presburger's paper from 1929 [159].

Difference logic was recognized as an interesting fragment of linear arithmetic by Pratt [158]. He considered "separation theory", which is the conjunctive fragment of what we call difference logic. He observed that most inequalities in verification conditions are of this form. Disjunctive difference logic was studied in M. Mahfoudh's PhD thesis [119] and in [120], among other places. A reduction of difference logic to SAT was studied in [187] (in this particular paper and some later papers, this theory fragment is called "separation logic", after Pratt's separation theory – not to be confused with the separation logic that is discussed in Chap. 8). The main reason for the renewed interest in this fragment is due to interest in **timed automata**: the verification conditions arising in this problem domain are difference logic formulas.

In general, the amount of research and writing on linear systems is immense, and in fact most universities offer courses dedicated to this subject. Most of the research was and still is conducted in the operations research community.

5.10 Glossary

The following symbols were used in this chapter:

Symbol	Refers to ...	First used on page ...
l_i, u_i	Constants bounding the i-th variable from below and above	113
m	The number of linear constraints in the original problem formulation	114
n	The number of variables in the original problem formulation	114
A	Coefficient matrix	115
$\mathbf{x}$	The vector of the variables in the original problem formulation	115
$\mathcal{B}, N$	The sets of basic and nonbasic variables, respectively	116

continued on next page

continued from previous page		
Symbol	**Refers to ...**	**First used on page ...**
α	A full assignment (to both basic and nonbasic variables)	116
θ	See (5.13)	118
β_i	Upper or lower bound	128
$\widehat{\mathrm{mod}}$	Symmetric modulo	131