5

Linear Arithmetic

5.1 Introduction

This chapter introduces decision procedures for conjunctions of linear constraints. Recall that this is all that is needed for solving the more general case in which there is an arbitrary Boolean structure, based on the algorithms that were described in Chap. 3.

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. The following example demonstrates how a compiler may use a decision procedure for arithmetic in order to optimize code.

Example 5.2. Consider the following C code fragment:

© Springer-Verlag Berlin Heidelberg 2016

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_5

```
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]`. A compiler might generate the assembly 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.

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, and they are relatively easy to implement in their basic form. 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

The Simplex algorithm, originally developed by Dantzig 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.

5.2.1 A Normal Form

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:[1]

$$l_i \le x_i \le u_i \; , \tag{5.4}$$

   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

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

m

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

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 5.3. Consider the following conjunction of constraints:

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

The problem is rewritten into the general form as follows:

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

▟

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.

[n]

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.[2] The (closed) 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 5.3 appears in Fig. 5.1.

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

[A]

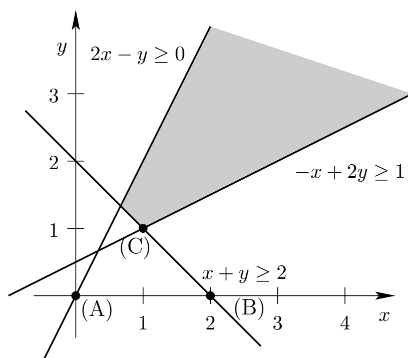

Fig. 5.1. A graphical representation of the problem in Example 5.3, projected on x and y. The shaded region corresponds to the set of satisfying assignments. The marked points (A), (B), and (C) illustrate the progress that the Simplex algorithm makes, as will be explained in the rest of this section

vector $\mathbf{x}$. Following this notation, our problem is equivalent to the existence of a vector $\mathbf{x}$ such that

x

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

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 5.4. We continue Example 5.3. 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 5.4 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

[2] 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.

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 submatrix 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 5.5. We continue our running example. The tableau and the bounds for Example 5.3 are

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

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

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 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:

Algorithm 5.2.1: GENERAL-SIMPLEX

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

- **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 5.6 (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 5.7. 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 [109]. 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 saw in Chap. 3.

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 5.8 (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, and "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 5.9. 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 [109], 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 discuss this property in detail in Sect. 11.6). 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. Thus, 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 [208]. The same bounds are applicable to the feasibility problem as well. Briefly, it was shown in [208] 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 problem. 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, 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, and hence can make branch and bound faster (this combination is known by the name **branch-and-cut**). Furthermore, if certain conditions are met—see Chap. 23.8 in [252] for details—Simplex + cutting planes of the type described below form a decision procedure for integer linear arithmetic.

Aside: BRANCH AND BOUND **for Integer Linear Programs**
When BRANCH AND BOUND is used for solving an optimization problem, after a first feasible solution is found, the search is continued until no smaller solution (assuming it is a minimization problem) can be found. A branch is pruned if the value of the objective according to a solution to the relaxed problem at its end is larger than the best solution found so far. The objective can also be used to guide the branching heuristic (which variable to split on next, and which side to explore first), e.g., find solutions that imply a small value of the objective function, so more future branches are pruned (bound) early.

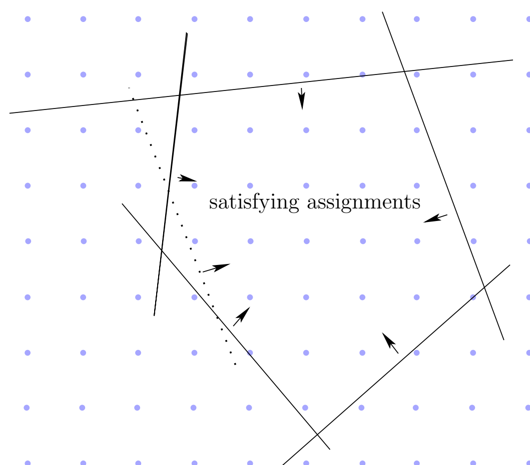

Fig. 5.2. The dots represent integer solutions. The thin dotted line represents a cutting-plane—a constraint that does not remove any integral solution

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 $1 \le x_1$ and $0.5 \le x_2$. 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 [109].

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:

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

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,

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

Equation (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

Similarly to the Simplex method (Sect. 5.2), the Fourier–Motzkin variable elimination algorithm takes a conjunction of linear constraints over real variables and decides their satisfiability. It is not as efficient as Simplex, but it can still be competitive for small formulas. In practice it is used mostly as a method for eliminating existential quantifiers, a topic that we will only cover later, in Sect. 9.2.4.

Let m denote the number of such constraints, and let $x_1, \ldots, x_n$ denote the variables used by these constraints. We begin by eliminating equalities.

5.4.1 Equality 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 5.10. 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

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

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

$$x \le y + 10,\ y \le 15,\ y \ge -x + 20\ . \tag{5.44}$$

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

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

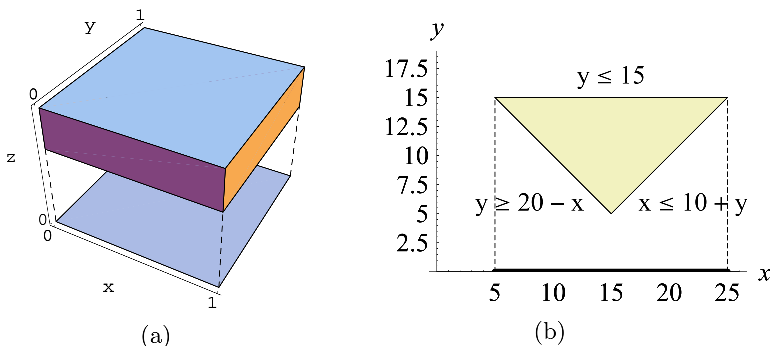

Fig. 5.3. Projection of constraints: (a) a cuboid is projected onto the x and y axes; (b) a triangle is projected onto the x axis

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:

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

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

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

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

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

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

$$\beta_l \leq x_n \leq \beta_u . \quad (5.49)$$

The following new constraint is added:

$$\beta_l \leq \beta_u . \quad (5.50)$$

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 5.11. Consider the following set of constraints:

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

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:

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

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.[3] 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, as a decision procedure, 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. It can be seen as a variant of the Fourier–Motzkin algorithm (Sect. 5.4). Both are not considered to be the fastest decision procedures, but they are used for existential quantifier elimination, a topic that will only be covered later, in Chap. 9.

Each conjunct is assumed to be either an equality of the form

[3] There are extensions of Simplex to strict inequalities. See, for example, [108].

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

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

The run time 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 5.12. 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 $1 \leq y$. ▟

As in the case of Fourier–Motzkin, 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.

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

$a \mathbin{\widehat{\mathrm{mod}}} b$

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

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

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

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:

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

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

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

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

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

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:

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

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 $a_i - (a_i \mathbin{\widehat{\bmod}} m)$ 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

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

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):

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

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 5.13. Consider the following formula:

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

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)):

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

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

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

Division by m results in

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

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,

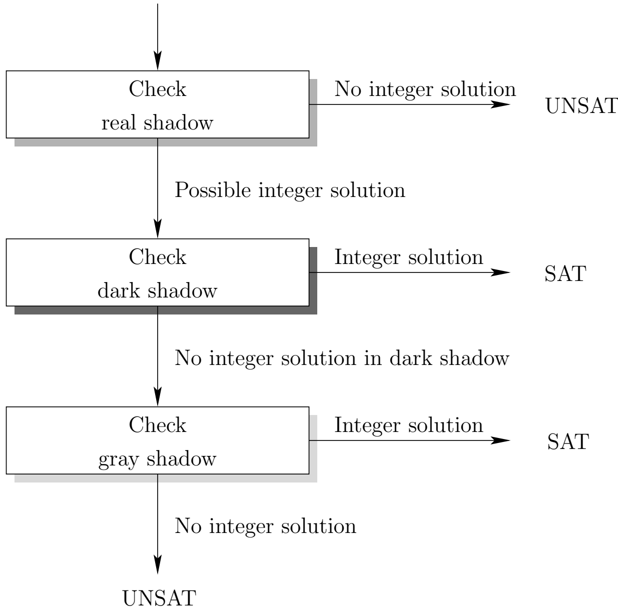

Fig. 5.4. Overview of the Omega test

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.

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:

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

The existence of such a variable z implies

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

Example 5.14. Consider the following set of constraints:

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

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. ▪

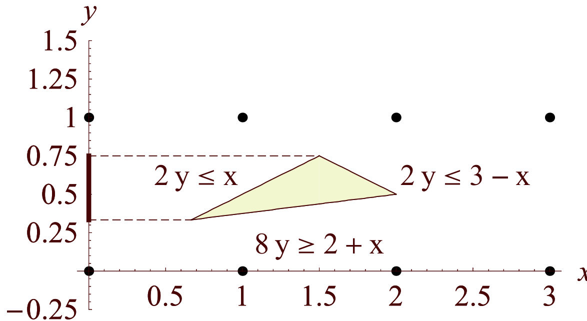

Fig. 5.5. Computing the real shadow: eliminating x

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.

Example 5.15. Consider the same set of constraints as in Example 5.14. 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

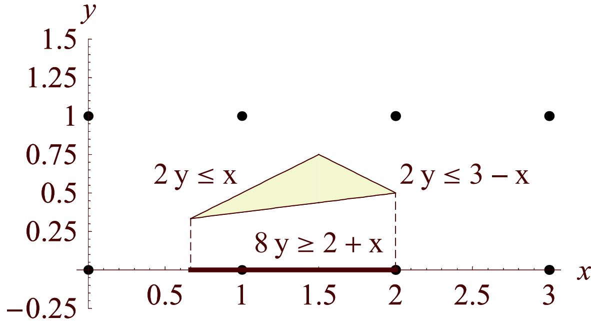

Fig. 5.6. Computing the real shadow: eliminating y

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.

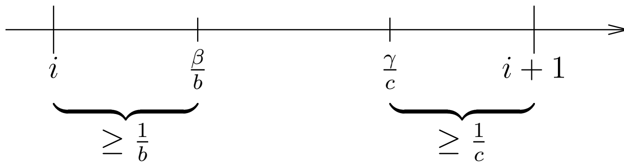

Fig. 5.7. Computing the dark shadow

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.[4]
2. After the previous preprocessing has been applied, strict inequalities can be transformed into nonstrict inequalities as follows:

[4] 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.

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

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.

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 5.16 (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 5.17. 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:

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

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

2. 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:

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

3. 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

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

The objective is to find a feasible 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. The Boolean structure is dealt with by DPLL(T), as described in Chap. 3.

Definition 5.18 (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 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 5.19. *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 5.18 and Theorem 5.19 to general difference logic (which includes both strict and nonstrict inequalities) is left as an exercise as well (see Problem 5.16).

By Theorem 5.19, deciding a difference logic formula amounts to searching for a negative cycle in a graph. This can be done with the **Bellman–Ford algorithm** [82] 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:

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

(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:

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

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.[5]

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?

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). Recall the definition $i = \lfloor \frac{\beta}{b} \rfloor$ in Sect. 5.5.3. Show that

- $\frac{\beta}{b} - i \geq \frac{1}{b}$, and
- $i + 1 - \dfrac{\gamma}{c} \geq \dfrac{1}{c}$

Recall that all coefficients are assumed to be integers.

Problem 5.12 (eliminating equalities). Show that

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

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:

[5] 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.

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

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 5.19.

Problem 5.16 (inequality graphs for difference logic). Extend Definition 5.18 and Theorem 5.19 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.

Problem 5.19 (theory propagation for difference logic). Recall the notion of exhaustive theory propagation that was studied in Sect. 3.4.2. Suggest an efficient procedure that performs exhaustive theory propagation for the case of difference logic (difference logic is presented in Sect. 5.7).

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. A somewhat more efficient way to eliminate variables called **virtual substitution** was suggested by Loos and Weispfenning [183].

The Simplex method was introduced by Dantzig in 1947 [84]. 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 [108] in the context of DPLL(T), a technique that we saw in Chap. 3. 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.

Consider, for example, how LP is used for computing an optimal placement of gates in an integrated circuit [152]. 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 [252]. Other resources on the subject that we found useful include publications by Wolsey [288], Hillier and Lieberman [142], and Vanderbei [278].

Gomory cutting planes are due to a paper published by Ralph Gomory in 1963 [134]. 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. [11]. The variant we described is suitable for working with the general Simplex algorithm, and its description here is based on [109].

The Omega test was introduced by Pugh as a method for deciding integer linear arithmetic within an optimizing compiler [233]. 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 [283]. Williams' work, in turn, is inspired by Presburger's paper from 1929 [232].

Difference logic was recognized as an interesting fragment of linear arithmetic by Pratt [231]. 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 [185] and in [186], among other places. A reduction of difference logic to SAT was studied in [267] (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	99
m	The number of linear constraints in the original problem formulation	99
n	The number of variables in the original problem formulation	100
A	Coefficient matrix	100
$\mathbf{x}$	The vector of the variables in the original problem formulation	101
$\mathcal{B}, \mathcal{N}$	The sets of basic and nonbasic variables, respectively	102
α	A full assignment (to both basic and nonbasic variables)	102
θ	See (5.13)	104
β_i	Upper or lower bound	114
$\widehat{\text{mod}}$	Symmetric modulo	117