Markus Grasmair

Discrete Optimisation

Lecture Notes
Winter 2010/11

Computational Science Center
University of Vienna
A-1090 Vienna, Austria

Preface

These notes are mostly based on the following book and lecture notes:

- Bernhard Korte and Jens Vygen, *Combinatorial Optimization*, Springer, 2000.
- Alexander Martin, *Diskrete Optimierung*, Technical University of Darmstadt, Germany, 2006.

The chapter on continuous linear optimisation in addition uses:

- Philippe G. Ciarlet, *Introduction to numerical linear algebra and optimisation*, Cambridge University Press, 1989.
- Otmar Scherzer and Frank Lenzen, *Optimierung*, Vorlesungsskriptum WS 2008/09, University of Innsbruck, Austria, 2009.

The examples are partly taken from:

- Lászlo B. Kovács, *Combinatorial Methods of Discrete Programming*, Akadémiai Kiadó, Budapest, 1980.
- Eugene L. Lawler, Jan K. Lenstra, Alexander H.G. Rinnooy Kan, David B. Shmoys, The Traveling Salesman Problem, John Wiley and Sons, 1985.

Markus Grasmair,
Vienna, 18th January 2011.

Contents

List of Algorithms

Chapter 1

Introduction

1.1 Classification of Optimisation Problems

Many problems in applications can be formulated aa *optimisation problems*, which consist in the minimisation of an *objective function* or *cost function* f over a set of *feasible variables* Ω. That is, one has to solve the problem:

$$\text{Minimise } f(x) \qquad \text{subject to } x \in \Omega\ .$$

Typically, the function f is defined on some larger set X, and the condition $x \in \Omega$ serves as an additional restriction of the set of solutions.

Depending on the function f and the sets X and Ω, one can make the following rough classification of optimisation problems. Each of the following classes requires a different approach for the actual computation of the optimum.

Discrete and Continuous Optimisation

Continuous Optimisation

Here the set X is a real vector space (typically, $X = \mathbb{R}^n$), and the set Ω is a continuous (that is, non-discrete) subset of X.

Discrete Optimisation

Here, either the set X or the set Ω is some discrete set. The typical situation is that of *integer pgrogramming*, where Ω is a subset of $\mathbb{Z}^n$. One also often uses the term *combinatorial optimisation* instead of discrete optimisation.

Binary Optimisation

This is the special case of discrete optimisation, where Ω is a subset of $\{0, 1\}^n$. Many problems in *graph theory* can be equivalently formulated as binary optimisation problems.

Mixed Integer Optimisation

Here $X = \mathbb{R}^n$, and Ω is some subset of $\mathbb{Z}^p \times \mathbb{R}^{n-p}$ with $1 \le p \le n - 1$. That is, part of the variables are integers, part are reals.

Restricted and Free Optimisation

In the case where $X = \mathbb{R}^n$, one has to differentiate between free and restricted optimisation:

Free Optimisation

Here $\Omega = \mathbb{R}^n$.

Restricted Optimisation

Here $\Omega \subsetneq \mathbb{R}^n$.

Typically, the set Ω is defined by a set of equalities and inequalities, that is,

$$\Omega = \big\{x \in \mathbb{R}^n : c_i^{(1)} = 0 \text{ for all } i \in I_1\big\} \cap \big\{x \in \mathbb{R}^n : c_i^{(2)}(x) \le 0 \text{ for all } i \in I_2\big\}$$

for some finite number of (continuous) functions $c_i^{(j)} \colon X \to \mathbb{R}$, $i \in I_j$, $j = 1, 2$.

Convex and Non-convex Optimisation

Again we assume that $X = \mathbb{R}^n$. Recall that a function $f \colon \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ is called *convex*, if

$$f\big(\lambda x + (1-\lambda)y\big) \le \lambda f(x) + (1-\lambda) f(y) \qquad \text{for all } x,\, y \in \mathbb{R}^n \text{ and } 0 \le \lambda \le 1\,.$$

In other words, the line connecting the points $(x, f(x))$ and $(y, f(y))$ lies always above the graph of f (that is, it is contained in the *epigraph* of f). Moreover, a set $K \subset \mathbb{R}^n$ is convex, if

$$\lambda x + (1-\lambda) y \in K \qquad \text{for all } x,\, y \in K \text{ and } 0 \le \lambda \le 1\,.$$

Note that the set K is convex, if and only if its *indicator function* χ_K, which is defined as $\chi_K(x) = 0$ for $x \in K$ and $\chi_k(x) = +\infty$ if $x \notin K$, is convex.

Convex Optimisation

In this case the objective function f is convex (and lower semi-continuous) and the set Ω is a convex (and closed) subset of $\mathbb{R}^n$. Typically, one has

$$\Omega = \big\{x \in \mathbb{R}^n : c_i(x) \le 0 \text{ for all } x \in I\big\}\,,$$

where $c_i \colon \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$, $i \in I$, are *convex* functions.

Quadratic Optimisation

Here the function $f \colon \mathbb{R}^n \to \mathbb{R}$ is quadratic, that is, there exist a (symmetric and positive semi-definite) matrix $G \in \mathbb{R}^n \times \mathbb{R}^n$, a vector $c \in \mathbb{R}^n$, and $d \in \mathbb{R}$ such that

$$f(x) = \frac{1}{2} x^T G x + c^T x + d\,.$$

Note that often we may assume without loss of generality that $d = 0$, as we are only interested in the minimiser of f and not the value at the minimiser.

Moreover, one often assumes that the constraints defining the set Ω of feasible variables are linear. That is, there exists a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $x \in \mathbb{R}^m$ such that

$$\Omega = \big\{x \in \mathbb{R}^n : Ax \le b\big\}\,.$$

Linear Optimisation

Here both the objective function as well as the constraints are linear. That is, there exist a vector $c \in \mathbb{R}^n$, a matrix $A \in \mathbb{R}^{m \times n}$, a vector $b \in \mathbb{R}^m$, and $d \in \mathbb{R}$ such that

$$f(x) = c^T x + d$$

and

$$\Omega = \big\{x \in \mathbb{K}^n : Ax \leq b\big\}$$

with either $\mathbb{K} = \mathbb{R}$ (continuous optimisation) or $\mathbb{K} = \mathbb{Z}$ (discrete optimisation). One often uses the term *linear programme* to denote a linear optimisation problem.

Somewhere between linear and quadratic optimisation, one finds *second order cone programmes*, where the objective function is linear, but some of the constraints are quadratic.

Non-convex Optimisation

Everything else.

Though—also here, one has to differentiate between *smooth* optimisation, where the objective functionals is (twice) differentiable, and *non-smooth* optimisation, which is still harder to treat.

Differences

In general, discrete optimisation problems are much harder to solve than continuous ones (at least, if the objective function has some structure like convexity or at least smoothness). In addition, free optimisation problems are easier to solve than restricted ones. Finally, there are huge differences between linear, convex, and non-convex optimisation problems. While linear problems can be solved exactly in finite time, most convex optimisation problems can only be solved approximately (using some kind of numerical approximation of the minimiser). In the case of non-convex problems, even the approximate minimisation is often not possible; the best one usually obtains are approximations of local minimisers.

Remarks

Often a concrete problem is not given as a minimisation problem, but rather as a maximisation problem. That is, one is given the task to maximise $f(x)$ subject to the constraint $x \in \Omega$. Such a problem can be easily reformulated as a minimisation problem by simply replacing the objective function f by $-f$.

Note that linear discrete optimisation as defined above also encompasses linear binary optimisation. Indeed, the constraint $x \in \{0,1\}^n$ can be equivalently written as $x \in \mathbb{Z}^n$, $x \leq 1$, $-x \leq 0$. Therefore, we may restrict ourselves to studying general discrete linear optimisation problems and need not differentiate between general discrete, and binary ones.

Note that every inequality constraint $Ax \geq b$ is equivalent to the opposite constraint $-Ax \leq -b$. In addition, an equality constraint $Ax = b$ holds if and only if the two inequality constraints $Ax \leq b$ and $-Ax \leq -b$ are satisfied simultaneously.

1.2 Examples

1.2.1 Knapsack Problem

Assume you have to pack your knapsack for a hiking tour (or your luggage for going on holidays). There are different items you can take with you, and each of them has its own use-value for you. In addition, every item has its weight, and you do not want to overload yourself (or you do not want to pay for overweight). What should you take with you, if you want to obtain the best use-value while still staying below the maximum possible (or permitted) weight?

More mathematically, you can formulate this problem as follows: Given a number n of objects o_j, $j = 1, \dots, n$, of respective weight $a_j > 0$ and use-value $c_j > 0$. The task is to choose a subset S of these objects in such a way that

$$\sum_{j\in S} c_j \text{ is maximal, while } \sum_{j\in S} a_j \le b$$

for some given maximal weight $b \ge 0$.

This problem can be quite easily rewritten as a binary linear optimisation problem. To that end we define the variable x_j as

$$x_j = \begin{cases} 1\,, & \text{if the object } o_j \text{ is chosen,} \\ 0\,, & \text{else.} \end{cases}$$

Then the problem is to maximise

$$f(x) := \sum_{j=1}^{n} c_j x_j$$

subject to the constraints

$$x_j \in \{0,1\} \text{ for all } 1 \le j \le n \qquad \text{and} \qquad \sum_{j=1}^{n} a_j x_j \le b\,.$$

A naive (greedy) approach for the solution of this problem would be to simply pack those items with the highest value-to-weight ratio. More precisely, we start with an empty knapsack and select from all those items that we still can carry one for which the ratio c_j/a_j is maximal. We add this item and repeat the procedure until the addition of any item would put us over the weight limit.

It is easy to see that this greedy approach often does not lead to an optimal solution. Consider, for instance the case where $n = 3$, $a_1 = 3$, $a_2 = a_3 = 2$, $c_1 = 5$, $c_2 = c_3 = 3$, and $b = 4$. Then

$$\frac{c_1}{a_1} = \frac{5}{3} > \frac{3}{2} = \frac{c_2}{a_2} = \frac{c_1}{a_1}\,.$$

Thus the greedy approach would consist in first packing the first item. Then we are already finished, as the addition of any other item is impossible. The total use value we obtain with this strategy is 5. The optimal solution, however, would be to pack the second and third item, in which case the total use value would be 6. This shows that the solution of this problem requires a refined approach.

1.2.2 Set Packing

Assume that we are given a finite set $E = \{1, \dots, m\}$ and a family F_j, $j \in \{1, \dots, n\}$, of subsets of E. In addition, to every subset F_j, a cost c_j is assigned. The task is to choose a subfamily of the F_j in such a way that the total cost is minimal, while every element of E is contained in at most (at least, exactly) one member of the subfamily.

This can be formulated as a binary programme as follows: We introduce the variable $x \in \{0,1\}^n$ and let $x_j = 1$ if we include the set F_j, and $x_j = 0$ else. In addition, we define a matrix $A \in \{0,1\}^{m\times n}$ the entries of which are defined as $a_{i,j} = 1$, if the element $i \in E$ is contained in F_j, and $a_{i,j} = 0$ else. Then the element $i \in E$ is contained in at least one of the subsets F_j that have been chosen, if $(Ax)_i = \sum_j a_{i,j}x_j \geq 1$. Similarly we can obtain the condition that every element has to be contained in at most or exactly one set, if we replace the sign $\geq$ by $\leq$ or $=$ in the inequality above.

We therefore arrive at the binary programme

$$c^T x \to \min \qquad \text{subject to } -Ax \leq 1,\ x \in \{0,1\}^n\ .$$

1.2.3 Assignment Problem

A company has n employees who are to be assigned n different tasks; each employee can perform precisely one task. The costs involved if employee i is assigned task j are $c_{i,j}$. The question is, how the tasks should be distributed among the employees in such a way that the total costs are minimal. Because of the restrictions that everybody has to perform precisely one job and every job has to be performed, we can write this as the optimisation problem

$$\sum_{i,\,j=1}^{n} x_{i,j}c_{i,j} \to \min$$

subject to

$$x_{i,j} \in \{0,1\}\,, \qquad \sum_{i=1}^{n} x_{i,j_0} = 1 = \sum_{j=1}^{n} x_{i_0,j} \qquad \text{for all } i_0,\, j_0 \in \{1,\dots,n\}\ .$$

A different way for interpreting the assignment problem is to formulate it as an optimisation problem on a graph. We consider a graph with vertices $v_i^{(e)}$, $1 \leq i \leq n$, (the employees) and $v_j^{(t)}$, $1 \leq j \leq n$, (the tasks) and an edge $e_{i,j}$ between each employee $v_i^{(e)}$ and each task $v_j^{(t)}$. Moreover, every edge $e_{i,j}$ is assigned the weight $c_{i,j}$. Because there is no edge between any two employees and, similarly, between any two tasks, the graph is bipartite with bipartition $(v_i^{(e)})_{1\leq i\leq n}\dot\cup(v_j^{(t)})_{1\leq j\leq n}$. The problem is now to find an *optimal matching* between the $v_i^{(e)}$ and the $v_j^{(t)}$, that is, to select edges $e_{i,j}$ in such a way that every vertex $v_i^{(e)}$ and every vertex $v_j^{(t)}$ belongs to precisely one of the selected edges, and the sum of the costs $c_{i,j}$ of the selected edges is minimal.

1.2.4 Traveling Salesman

A traveling salesman has to visit n cities and in the end return to the city from which he started. The question is, in which order he should plan the visits such that the total cost of his travels is minimal. Here it is assumed that the costs for traveling between any pair of cities is given.

In terms of graph theory, we are given a complete graph G with vertex set consisting of the n cities (undirected or directed, depending on the question whether the cost of traveling between any two cities does depend on the direction of the voyage). In addition, we have a cost functional c on the set of all edges. The task is to solve the optimisation problem

$$c(C) \to \min \qquad \text{such that } C \text{ is a Hamiltonian circuit in } G.$$

In the following, we show how this problem can be formulated as a binary linear programme. To that end let $c_{i,j}$ denote the cost for traveling from city i to city j. Moreover we introduce the discrete variables $x_{i,j} \in \{0,1\}$, where $x_{i,j} = 1$ if a travel from i to j is included in the tour and $x_{i,j} = 0$ else. Then the total cost of the travel is

$$\sum_{i,\,j=1}^{n} c_{i,j} x_{i,j}\ .$$

The objective functional is therefore easy to encode. For the constraint, namely that the travel should be a tour including every city precisely once, the situation is more complicated. The easy part is to encode the constraint that every city is visited exactly once. This can be seen by observing the trivial fact that in the above notation this is equivalent to stating that every city is entered precisely once and also left again precisely once. This leads to the constraints

$$\begin{aligned} \sum_{i=1}^{n} x_{i,j} &= 1 \qquad \text{for all } 1 \le j \le n\,, \\ \sum_{j=1}^{n} x_{i,j} &= 1 \qquad \text{for all } 1 \le i \le n\ . \end{aligned} \tag{1.1}$$

Until now, the problem is precisely the same as the assignment problem. We have, however, not yet included the assumption that we have to make a round trip; as for now the constraints allow for several disconnected round trips, including trivial cases where $c_{i,j} = 1$ for some i, that is, we leave the city i only to return to it at the same moment.

There are several possibilities to exclude smaller round trips from the set of feasible solutions. To that end let for the moment $x := (x_{i,j})_{i,j} \in \{0,1\}^{n\times n}$ satisfying (1.1) be fixed. Let $I \subsetneq \{1,\dots,n\}$ be some subset of the cities we have to visit. If the travel defined by x does not include a round trip through the cities in I, then we must leave the set I at least once. That is, there exists some $i \in I$ and $j \notin I$ such that $x_{i,j} = 1$. Therefore

$$\sum_{i\in I}\sum_{j\notin I} x_{i,j} \ge 1\ .$$

Because of (1.1), this is equivalent to the inequality

$$\sum_{i,\,j\in I} x_{i,j} \le |I| - 1\ . \tag{1.2}$$

Indeed, the requirement that (1.2) holds for every $I \subsetneq \{1, \ldots, n\}$ is equivalent to the non-existence of subtours. Thus, we have shown that the traveling salesman problem can be brought into the form of a discrete linear programme. The formulation adopted here, however, is of no use for any practical purpose: The number of inequalities we have added equals the number of proper non-empty subsets of $\{1, \ldots, n\}$, that is, $2^n - 2$.

It is possible to formulate the constraints above in such a way that we only add $(n-1)^2$ additional inequalities to (1.1), but instead add $n-1$ new variables $u_i \in \mathbb{R}$, $2 \leq i \leq n$, which do not appear in the cost functional. One can show that the exclusion of smaller round trips is equivalent to the set of inequalities

$$u_i - u_j + n x_{i,j} \leq n - 1 \qquad \text{for all } i,\, j \in \{2, \ldots, n\} \;.$$

In constrast to the formulation above, this leads to a mixed integer programme with n^2 binary variables and $n - 1$ real variables.

Chapter 2

Linear Programming

2.1 Polyhedra

In this chapter we describe the simplex algorithm, which can be used for solving linear programmes of the form

$$c^T x \to \min \qquad \text{subject to } Ax \leq b\,. \tag{2.1}$$

Here $c \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$. The inequality $Ax \leq b$ is interpreted componentwise, that is,

$$Ax \leq b \qquad \text{if and only if} \qquad (Ax)_i \leq b_i \text{ for every } 1 \leq i \leq m\,.$$

Before turning to the algorithmical solution of (2.1), we first study the problem more closely.

Definition 2.1.1. Let $P \subset \mathbb{R}^{m\times n}$. The set P is a *polyhedron*, if there exist $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ such that

$$P = \big\{x \in \mathbb{R}^n : Ax \leq b\big\}\,.$$

If the polyhedron P is bounded, then it is called a *polytope*.

Conversely, let $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ be given. We define

$$\mathcal{P}(A,b) := \big\{x \in \mathbb{R}^n : Ax \leq b\big\}$$

the *polyhedron* defined by $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$. ■

Thus, the problem (2.1) can be equivalently formulated as

$$c^T x \to \min \qquad \text{subject to } x \in \mathcal{P}(A,b)\,.$$

The main difference between the two formulations is that in the latter case the matrix A and the vector b are not given explicitely, but only the set that they describe.

Lemma 2.1.2. *Consider the linear programme* $c^T x \to \min$ *subject to* $x \in \mathcal{P}(A,b)$*. Then there are three possibilities:*

1. *The set $\mathcal{P}(A,b)$ is empty, that is, there exist no feasible variables.*
2. *The objective function $x \mapsto c^T x$ has no lower bound on $\mathcal{P}(A,b)$.*
3. *The linear programme admits a solution.*

The importance of this result can be grasped from only slightly more complicated optimisation problems, where it does not hold anymore. Consider for instance the quadratic programme of minimizing the linear objective function $(x_1, x_2) \mapsto x_1$ subject to the constraints $x_1 \geq 0$ and $x_1 x_2 \geq 1$. Here the admissible set is non-empty and the objective function is non-negative on this set. Still, the problem admits no solution, because the second variable tends to infinity if the first approaches zero.

An additional important property of linear programmes is that solutions (if existent) can always be found at the boundary of $\mathcal{P}(A,b)$. Thus, for the search for solutions, we may restrict ourselves to the boundary of the polyhedron. In the following we will see that even further restrictions are possible.

Definition 2.1.3. Let $\mathcal{P}(A,b)$ be a non-empty polyhedron and let $\xi \in \mathbb{R}^n$. Assume that

$$d := \max\bigl\{\xi^T x : x \in \mathcal{P}(A,b)\bigr\}$$

is finite. Then the set

$$H(\xi, d) := \bigl\{x \in \mathbb{R}^n : \xi^T x = d\bigr\}$$

is called a *supporting hyperplane* of the polyhedron $\mathcal{P}(A,b)$.

A non-empty set $F \subset \mathbb{R}^n$ is called a *face* of the polyhedron $\mathcal{P}(A,b)$ if either $F = \mathcal{P}(A,b)$, or there exists a supporting hyperplane $H(\xi,d)$ of $\mathcal{P}(A,b)$ such that $F = H(\xi,d) \cap \mathcal{P}(A,b)$. If $x \in \mathbb{R}^n$ is such that $\{x\}$ is a face of $\mathcal{P}(A,b)$, then x is called a *vertex* of $\mathcal{P}(A,b)$. The vertices of $\mathcal{P}(A,b)$ are also called the *basic solutions* of the system of inequalities $Ax \leq b$.

A face F of $\mathcal{P}(A,b)$ is called *minimal*, if it contains no other face of $\mathcal{P}(A,b)$. In particular, every vertex is a minimal face. ■

Proposition 2.1.4. *A non-empty set $F \subset \mathbb{R}^n$ is a minimal face of the polyhedron $\mathcal{P}(A,b)$, if and only if $Ax \leq b$ for all $x \in F$ (that is, $F \subset \mathcal{P}(A,b)$), and there exists a set of indices $B \subset \{1,\ldots,m\}$ such that*

$$F = \bigl\{x \in \mathbb{R}^n : (Ax)_i = b_i \textit{ for all } i \in B\bigr\}\ .$$

Corollary 2.1.5. *All minimal faces of the polyhedron $\mathcal{P}(A,b)$ with $0 \neq A \in \mathbb{R}^{m\times n}$ are affine subspace of $\mathbb{R}^n$ (note that 0-dimensional subspaces are sets containing a single point). Moreover, the dimension of all minimal faces is $n - \operatorname{rank} A$. If A is a polytope, then all minimal faces are vertices.*

Lemma 2.1.6. *Let $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$. If the linear programme $c^T \to \min$ subject to $Ax \leq b$ admits a solution, then there exists some minimal face F of $\mathcal{P}(A,b)$ such that every $x \in F$ is a solution.*

In particular, if the polyhedron $\mathcal{P}(A,b)$ has any vertex (that is, the matrix A has rank n), then there exists a vertex that solves the linear programme.

Note that the previous lemma does not claim that *every* solution of a linear programme is contained in a minimal face. Nevertheless, it allows us to restrict the search for minimisers to the set of minimal faces. In addition, Proposition 2.1.4 implies that we can find all minimal faces of $\mathcal{P}(A,b)$ by solving subsystems of the equation $Ax = b$. Since the number of subsystems of $Ax = b$ is finite, it follows that linear programmes can be solved in finite time by simply computing all minimal faces (which is a tedious task) and minimising the objective function $x \mapsto c^T x$ on these faces (which is easy). In addition, one has to check whether the objective functional is unbounded below on the feasible polyhedron $\mathcal{P}(A,b)$. While this brute force strategy leads to a finite algorithm, it is far from being efficient. A better method is the simplex algorithm, which will be discussed in the next sections.

2.2 Canonical Form of Linear Programmes

Definition 2.2.1. A linear optimisation problem is in *canonical form*, if it reads as

$$c^T x \to \min \qquad \text{subject to } Ax = b \text{ and } x \geq 0\ .$$

■

Remark 2.2.2. Consider the linear programme

$$c^T x \to \min \qquad \text{subject to } Ax \leq b$$

with $c \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$.

In the following, we will bring this linear programme into canonical form by introducing additional *slack variables*, that is, variables that do not influence the cost functional, and enlarging the vector c and the matrix A.

To that end note first that the inequality constraint $Ax \leq b$ can be written as an equality constraint by introducing the slack variable $y \in \mathbb{R}^n$ and noting that

$$Ax \leq b \qquad \text{if and only if } Ax + y = b \text{ and } y \geq 0\ .$$

In addition, we have to introduce a non-negativity constraint for all the occurring variables. This can be achieved by writing $x = \hat{x} - \tilde{x}$ with $\hat{x} \geq 0$ and $\tilde{x} \geq 0$, and noting that

$$\begin{aligned} c^T x &= c^T(\hat{x} - \tilde{x}) = c^T\hat{x} - c^T\tilde{x}\,, \\ Ax &= A(\hat{x} - \tilde{x}) = A\hat{x} - A\tilde{x}\ . \end{aligned}$$

Thus we arrive at the (enlarged) linear programme in canonical form

$$(c^T, -c^T, 0)\begin{pmatrix}\hat{x}\\ \tilde{x}\\ y\end{pmatrix} \to \min$$

subject to

$$(A, -A, \mathrm{Id})\begin{pmatrix}\hat{x}\\ \tilde{x}\\ y\end{pmatrix} = b \qquad \text{and} \qquad (\hat{x}, \tilde{x}, y) \geq 0\ .$$

Note that the slack variables influence only the constraint, but not the cost functional. ■

Example 2.2.3. Consider the set of inequalities

$$\begin{aligned} x_1 - x_2 &\le 1\,, \\ -2x_1 - x_2 &\le 2\,, \\ x_2 &\le 1\,. \end{aligned}$$

In matrix form, these read as

$$\begin{pmatrix} 1 & -1 \\ -2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \le \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\,.$$

Introducing the slack variable $y \in \mathbb{R}^3$, $y \ge 0$, we obtain

$$\begin{pmatrix} 1 & -1 & 1 & 0 & 0 \\ -2 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\,, \qquad y_i \ge 0\,.$$

Finally, we add the non-negativity constraint for x by writing $x = \hat{x} - \tilde{x}$. Then we obtain the canonical form

$$\begin{pmatrix} 1 & -1 & -1 & 1 & 1 & 0 & 0 \\ -2 & -1 & 2 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \tilde{x}_1 \\ \tilde{x}_2 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\,, \quad \hat{x}_j \ge 0\,,\ \tilde{x}_j \ge 0\,,\ y_i \ge 0\,.$$

■

Remark 2.2.4. An important feature of the method of normalisation described in Remark 2.2.2 is the fact that the linear programmes in canonical forms it yields have matrices of full rank, and therefore the minimal faces of the feasible polyhedron (if non-empty) are vertices. In addition, if $b \ge 0$, then one vertex is the point $\hat{x} = 0$, $\tilde{x} = 0$, $y = b$. ■

2.3 The Simplex Algorithm

We assume in the following that we are given a linear programme in canonical form, where the matrix $A \in \mathbb{R}^{m\times n}$ has full rank and $n > m$ (these assumptions can be satisfied because of Remark 2.2.2). That is, we want to solve the problem

$$c^T x \to \min \qquad \text{subject to } Ax = b \text{ and } x \ge 0\,, \tag{2.2}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $n > m$, and $\operatorname{rank}(A) = m$.

In addition, we assume for the moment that the feasible polyhedron $\{x \in \mathbb{R}^n : Ax = b,\ x \ge 0\}$ is non-empty, which implies that the feasible polyhedron has some vertex. We will show later, how we can decide whether this assumption is satisfied. More precisely, we will describe an algorithm (again the simplex algorithm, but for a modified problem) that decides whether the feasible polyhedron is empty and, in case it is not, constructs a vertex, which we can then use for the initialisation of the simplex algorithm for the original problem.

Definition 2.3.1. Let $B \subset \{1, \dots, n\}$ be some subset with $|B| = m$. Define

$$A_B := (a_{ij})_{1 \le i \le m,\, j \in B}$$

the $m \times m$ submatrix of A generated by the indices B.

The matrix A_B is called *feasible basis*, if A_B is regular (that is, $\det A_B \neq 0$), and we have

$$x^{(B)} := A_B^{-1} b \ge 0 \ .$$

In this case, $x^{(B)} \in \mathbb{R}^m$ is called the *basic vector corresponding to A_B*. ■

Lemma 2.3.2. *Let $B \subset \{1, \dots, n\}$ be such that A_B is a feasible basis, and let $x^{(B)} \in \mathbb{R}^m$ be the corresponding basic vector. Let moreover $\tilde{x}^{(B)} \in \mathbb{R}^m$ be such that $(\tilde{x}^{(B)})_{j \in B} = x^{(B)}$ and $(\tilde{x}^{(B)})_{j \in B'} = 0$ (that is, we obtain $\tilde{x}^{(B)}$ from $x^{(B)}$ by inserting zeros at those indices not contained in B. Then $\tilde{x}^{(B)}$ is a basic solution of the system $Ax = b$, $x \ge 0$.*

Conversely, for every basic solution x of the system $Ax = b$, $x \ge 0$, there exists a feasible basis A_B with $B \subset \{1, \dots, n\}$ such that $x = \tilde{x}^{(B)}$.

Remark 2.3.3. Let $x^{(B)}$ be the basic vector corresponding to some feasible basis A_B. The vector $x^{(B)}$ is called *degenerate*, if $x_i^{(B)} = 0$ for some $i \in B$. In this case we can replace this index i by any index $j \in B'$ and still obtain the same basic vector. The possible existence of degenerate basic solutions leads to some complications in the solution of linear programmes when using the simplex algorithm. ■

For the next result, we introduce some more notation. We define, for given $B \subset \{1, \dots, n\}$,

$$B' := \{1, \dots, n\} \setminus B \qquad \text{and} \qquad A_{B'} := (a_{ij})_{1 \le i \le m,\, j \in B'} \ .$$

Moreover, we let

$$c_B := (c_j)_{j \in B} \qquad \text{and} \qquad c_{B'} := (c_j)_{j \in B'} \ .$$

Assume now that $x^{(B)} \in \mathbb{R}^m$ is a basic vector corresponding to some set $B \subset \{1, \dots, n\}$ and that $x^{(B)}$ is non-degenerate (that is, $x_i^{(B)} > 0$ for all $i \in B$). Choose some index $j_0 \in B'$ and consider the vector $z^{(t)} \in \mathbb{R}^{B'}$ satisfying $z_{j_0} = t$ and $z_j = 0$ for $j \neq j_0$. Next, we want to define entries z_i, $i \in B$, in such a way that the vector $z^{(t)}$ becomes admissible for some $t > 0$. The relation $Az^{(t)} = b$ implies that

$$x^{(B)} = A_B^{-1} b = A_B^{-1} A z^{(t)} = z_B^{(t)} + t(A_B^{-1} A)_{\cdot, j_0} \tag{2.3}$$

with $z_B^{(t)}$ denoting the vector corresponding to the components of $z^{(t)}$ lying in the set B. Because by assumption $x^{(B)} > 0$, it follows that, for $t > 0$ small enough, the vector $z^{(t)}$ also satisfies $z^{(t)} \ge 0$ and therefore is feasible for the linear programme. Moreover, the cost of the vector $z^{(t)}$ equals

$$c^T z^{(t)} = c_B^T x^{(B)} - t(c_B^T (A_B^{-1} A)_{\cdot, j_0} - c_{j_0}) \ .$$

As $t > 0$, the costs are smaller than those of the vector $\tilde{x}^{(B)}$ if and only if

$$c_{j_0} - c_B^T (A_B^{-1} A)_{\cdot, j_0} < 0 \,.$$

If this inequality holds, it makes sense to replace the vector $\tilde{x}^{(B)}$ by $z^{(t)}$. In addition, we should choose t as large as possible, that is, we set

$$t := \max\bigl\{ s > 0 : z^{(s)} \text{ is admissible } \bigr\} \,.$$

This maximum can easily be computed from equation (2.3). Moreover, if this choice of t is well-defined (that is, $t \neq +\infty$), then the vector $z^{(t)}$ will again be a basic solution.

These observations form the basis of the simplex algorithm, which is described in the following theorem. There, also the cases $t = +\infty$ and $x^{(B)}$ degenerate are treated.

Theorem 2.3.4. *Assume that $B \subset \{1, \ldots, n\}$ is such that A_B is a feasible basis, and let $\tilde{x}^{(B)}$ be the corresponding basic solution according to Lemma 2.3.2. Then the following hold:*

1. *If*
$$c_{B'}^T - c_B^T A_B^{-1} A_{B'} \geq 0 \,,$$
then $\tilde{x}^{(B)}$ is a solution of (2.2).

2. *If there exists $j_0 \in B'$ such that*
$$(c_{B'})_{j_0} - (c_B^T A_B^{-1} A_{B'})_{j_0} < 0 \,, \qquad \text{and} \qquad (A_B^{-1} A_{B'})_{\cdot, j_0} \leq 0 \,,$$
then the problem (2.2) *is unbounded.*

3. *Assume that there exist $j_0 \in B'$ and $\tilde{\imath} \in B$ such that*
$$(c_{B'})_{j_0} - (c_B^T A_B^{-1} A_{B'})_{j_0} < 0 \,, \qquad \text{and} \qquad \tilde{x}_{\tilde{\imath}}^{(B)} > 0 \,,$$
and for all $i \in B$ we have
$$\tilde{x}_i^{(B)} > 0 \qquad \text{whenever} \qquad (A_B^{-1} A_{B'})_{i, j_0} > 0 \,.$$
Define
$$\lambda := \min \biggl\{ \frac{\tilde{x}_i^{(B)}}{(A_B^{-1} A_{B'})_{i, j_0}} : i \in B \text{ and } (A_B^{-1} A_{B'})_{i, j_0} > 0 \biggr\} \tag{2.4}$$
and
$$y := \begin{cases} \tilde{x}_i^{(B)} - \lambda (A_B^{-1} A_{B'})_{i, j_0} \,, & \text{if } i \in B \,, \\ \lambda \,, & \text{if } i = j_0 \,, \\ 0 \,, & \text{if } i \in B' \setminus \{j_0\} \,, \end{cases}$$
Let moreover $i_0 \in B$ be any index for which the minimum in (2.4) *is attained.*

   Then $C := (B \setminus \{i_0\}) \cup \{j_0\}$ is a feasible basis and $y = \tilde{x}^{(C)}$ is the basic solution corresponding to C. Moreover $c^T \tilde{x}^{(C)} < c^T \tilde{x}^{(B)}$.

4. *Assume that there exists some* $j \in B'$ *with* $(c_{B'})_j - (c_B^T A_B^{-1} A_{B'})_j < 0$. *Assume moreover that for every* $j \in B'$ *with*
$$(c_{B'})_j - (c_B^T A_B^{-1} A_{B'})_j < 0 \tag{2.5}$$
there exists $i \in B$ *with*
$$(A_B^{-1} A_{B'})_{i,j} > 0 \qquad \text{and} \qquad \tilde{x}_i^{(B)} = 0\ . \tag{2.6}$$
Let $j_0 \in B'$ *and* $i_0 \in B$ *be such that* (2.5) *and* (2.6) *are satisfied. Define* $C := (B \setminus \{i_0\}) \cup \{j_0\}$. *Then*
$$\tilde{x}^{(C)} = \tilde{x}^{(B)}\ .$$

This last theorem can be used for defining an iterative algorithm for the solution of the linear programme (2.2). To that end one starts with some feasible basis A_B corresponding to some set $B \subset \{1, \ldots, n\}$ (we will discuss below, how we can obtain one) and then checks, which of the cases of Theorem 2.3.4 applies.

If we are in the case 1, then we have found a solution. Else, we are in at least one of the cases 2–4 (note that 2 and 3 can occur simultaneously). If we are in case 2, then the problem admits no solution, and we are done. If we are in cases 3 or 4, then we change the set B according to the strategy defined there. In both cases, we remove one index from B and replace it with an index in B'. Then we repeat this check with the updated set B.

Because the objective functional decreases each time we are in case 3, and there are only finitely many vertices to be visited during the algorithm, it follows that case 3 can only occur finitely many times. Thus there are three possibilities for the algorithm:

1. After a finite occurrence of the cases 3 and/or 4, we finally are in situation 1. Then the last vertex is a solution of the linear programme.

2. After a finite occurrence of the cases 3 and/or 4, we finally are in situation 2. Then the problem has no solution.

3. After a finite occurrence of the cases 3 and/or 4, we arrive at a non-terminating sequence of the case 4. That is, after a finite number of changes of the basis B in case 4, we return to a basis already chosen in a previous step. This phenomenon is known as *cycling*. In this case, the algorithm does not terminate.

Remark 2.3.5. It is possible to avoid cycling, if one chooses the indices i_0 and j_0 in cases 3 and 4 according to a suitable strategy. The easiest to implement is Bland's pivoting rule. Here, one defines j_0 as
$$j_0 = \min\bigl\{j : (c_{B'})_j - (c_B^T A_B^{-1} A_{B'})_j < 0\bigr\}\,,$$
and i_0 as
$$i_0 = \min\Bigl\{i : \frac{\tilde{x}_i^{(B)}}{(A_B^{-1} A_{B'})_{i,j_0}} = \lambda\Bigr\}\,,$$
where
$$\lambda = \min\biggl\{\frac{\tilde{x}_i^{(B)}}{(A_B^{-1} A_{B'})_{i,j_0}} : i \in B \text{ and } (A_B^{-1} A_{B'})_{i,j_0} > 0\biggr\}\ .$$
In other words, j_0 and i_0 are always minimal. ■

Remark 2.3.6 (Simplex tableau). The state of the simplex algorithm can be represented in the *simplex tableau*, which is the array

$$\begin{array}{c|cc} & & B' \\ \hline & c_B^T A_B^{-1} b & -c_{B'}^T + c_B^T A_B^{-1} A_{B'} \\ B & A_B^{-1} b & A_B^{-1} A_{B'} \end{array}$$

This tableau contains all necessary information used in the simplex algorithm. In particular, the basic solution $x^{(B)}$ corresponding to B and also the value of the cost functional at $x^{(B)}$ can be read off the tableau, as

$$\tilde{x}^{(B)} = A_B^{-1} b \qquad \text{and} \qquad c^T x^{(B)} = c_B^T A_B^{-1} b\,. \qquad \blacksquare$$

Remark 2.3.7. In each step of the simplex algorithm, in order to update the simplex tableau, one has to compute the matrices and vectors $A_B^{-1} A_{B'}$, $A_B^{-1} b$, $c_{B'}^T - c_B^T A_B^{-1} A_{B'}$, and $c_B^T A_B^{-1} b$. Because only two indices in the set B are changed in each iteration, these can be easily computed from the simplex tableau of the previous step. The required computations are summarised in Algorithm 1. $\blacksquare$

The simplex algorithm needs as input a set of indices $B \subset \{1, \ldots, n\}$ such that A_B is a feasible basis. In the following, we show how we such a set can be obtained. First we note that we can assume without loss of generality that $b \geq 0$. Indeed, if $b_{i_0} < 0$, then we can simply multiply the i_0-th line in A with -1, that is, we replace $a_{i_0,j}$ by $-a_{i_0,j}$ for all j.

Consider therefore the canonical linear programme $c^T x \to \min$ subject to $Ax = b$ and $x \geq 0$, and assume that $b \geq 0$. Then we can define an auxiliary programme

$$\mathbb{1}^T y \to \min \qquad \text{subject to } Ax + y = b,\ x \geq 0,\ y \geq 0\,, \tag{2.7}$$

where $\mathbb{1} \in \mathbb{R}^m$ is the m-dimensional vector whose entries are all equal to 1. In other words, the objective functional is simply the sum of the entries of the variable y. Because of the constraint $y \geq 0$, it follows that $\mathbb{1}^T y \geq 0$ for all admissible y, and $\mathbb{1}^T y = 0$, if and only if $y = 0$. Moreover the constraints $Ax + y = b$ and $x \geq 0$ show that $y = 0$ if and only if the equation $Ax = b$ holds for some $x \geq 0$. Thus, the minimal value of the auxiliary programme (2.7) equals 0, if and only if the original problem is feasible. Now let (x_0, y_0) be a basic solution of (2.7). If $\mathbb{1}^T y_0 = 0$, then also $y_0 = 0$ and thus the only non-zero components appear in the variable x_0. Because by assumption (x_0, y_0) is basic and x_0 satisfies $Ax_0 = b$ and $x_0 \geq 0$, it follows that x_0 is also a basic solution of the original problem, and therefore x_0 a vertex, which we can use as input for the simplex algorithm.

The main point in introducing the auxiliary problem (2.7) is that this problem is feasible and has an easy to find feasible basis. Indeed, the point $(x, y) = (0, b)$ is a basic solution and the corresponding feasible basis is simply the identity matrix (corresponding to the y-variable). Thus we can use the simplex algorithm for solving (2.7), and then, if the minimal value of (2.7) turns out to be zero, use the result as input for the simplex algorithm for the solution of the original problem. Moreover, if the solution of the auxiliary problem is non-degenerate, then it corresponds to a feasible basis, which only contains indices

```
Data: $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ with $m < n$, $b \in \mathbb{R}^m$;
Input: $B = (j_1, \dots, j_m)$ such that $A_B$ is a feasible basis;
Result: Either a solution $x$ of the linear programme $c^T x \to \min$ subject
        to $Ax = b$ and $x \geq 0$ or the knowledge that the linear
        programme is unbounded;
Initialisation: Set $A_B := (a_{i,j_k})_{i,k}$, let $B' = (j'_1, \dots, j'_{m-n})$ such that
                $B \cup B' = \{1, \dots, n\}$, and $A_{B'} := (a_{i,j'_k})_{i,k}$. Denote
                $c_B := (c_{j_k})_k$ and $c_{B'} := (c_{j'_k})_k$. Compute
                $V = (v_{i,k})_{0 \leq i \leq m,\, 0 \leq k \leq n-m}$ setting

                $$\begin{aligned} v_{0,0} &:= c_B^T A_B^{-1} b\,, & \\ v_{0,k} &:= -c_k + (c_B^T A_B^{-1} A_{B'})_k\,, & \text{for } 1 \leq k \leq m, \\ v_{i,0} &:= (A_B^{-1} b)_i\,, & \text{for } 1 \leq i \leq m, \\ v_{i,k} &:= (A_B^{-1} A_{B'})_{i,k}\,, & \text{for } 1 \leq i, k \leq m. \end{aligned}$$

while there exists $1 \leq k \leq n - m$ with $v_{0,k} < 0$ do
    Let $k_0 = \arg\min_k \{ j'_k : v_{0,k} < 0 \}$;
    if $v_{i,k_0} \leq 0$ for all $1 \leq i \leq m$ then
        there exists no solution;
        break;
    else
        Define
            $$\lambda = \min\left\{ \frac{v_{i,0}}{v_{i,k_0}} : 1 \leq i \leq m \text{ and } v_{i,k_0} > 0 \right\},$$
            $$i_0 = \arg\min_i \left\{ j_i : \frac{v_{i,0}}{v_{i,k_0}} = \lambda \right\},$$
        replace $j_{i_0} \leftarrow k_0$ and $j'_{k_0} \leftarrow i_0$ and set

            $$v_{i,k} \leftarrow v_{i,k} - \frac{v_{i_0,k} v_{i,k_0}}{v_{i_0,k_0}} \quad \text{for } \begin{array}{l} 0 \leq i \leq m,\ i \neq i_0\,, \\ 0 \leq k \leq n-m,\ k \neq k_0\,, \end{array}$$
            $$v_{i,k_0} \leftarrow \frac{v_{i,k_0}}{v_{i_0,k_0}} \quad \text{for } 1 \leq i \leq m,\ i \neq i_0\,,$$
            $$v_{i_0,k} \leftarrow -\frac{v_{i_0,k}}{v_{i_0,k_0}} \quad \text{for } 1 \leq k \leq n-m,\ k \neq k_0\,,$$
            $$v_{i_0,k_0} \leftarrow \frac{1}{v_{i_0,k_0}}.$$
    end
end
Define $x := 0 \in \mathbb{R}^n$;
foreach $i = 1, \dots, m$ do
    $x_{j_i} \leftarrow v_{i,0}$;
end
```

Algorithm 1: Simplex algorithm

corresponding to the x-variable. This feasible basis is then also feasible for the original problem and can be used for the initialisation of its simplex tableau.

Remark 2.3.8. Assume that we are given a linear programme of the form $c^T x \to \min$ subject to $Ax \le b$, and that $b \ge 0$. Then the problem is feasible, because $0 \in \mathcal{P}(A,b)$. If we now use the method described in Remark 2.2.2 for bringing the programme in canonical form, then we obtain the programme

$$(c^T, -c^T, 0) \begin{pmatrix} \hat{x} \\ \tilde{x} \\ y \end{pmatrix} \to \min$$

subject to

$$(A, -A, \mathrm{Id}) \begin{pmatrix} \hat{x} \\ \tilde{x} \\ y \end{pmatrix} = b \qquad \text{and} \qquad (\hat{x}, \tilde{x}, y) \ge 0 \; .$$

Then $y = b$ is a vertex and, again, the corresponding feasible basis is just the identity matrix for the variable y. The initial simplex tableau is then

$$\begin{array}{c|cc} & \multicolumn{2}{c}{B'} \\ \hline & 0 & (c^T, -c^T) \\ B & b & (A, -A) \end{array}$$

with B indicating the indices corresponding to y and B' the indices corresponding to $\hat{x}$ and $\tilde{x}$. ■

2.4 Remarks

In theory, the simplex algorithm is far from being efficient. In fact, it has an exponential complexity, and it is possible to construct examples of linear programmes with n variables and $2n$ constraints, where the simplex algorithm (with the pivoting rule introduced above) takes 2^n iterations. In practice, however, the simplex algorithm works quite fast. Moreover it has been subject of much research, and there exist many methods for speeding up the algorithm by using different, refined pivoting rules (that is, rules for the selection of the indices i_0 and j_0 in the updating steps).

On the other hand, there exists also a polynomial time algorithm for linear programming, the *ellipsoid method.* Being polynomial, one might expect that this method should work better than the simplex algorithm. In practice, however, it does not. First, it is difficult to implement (in contrast to the simplex algorithm), and, second, although polynomial, the complexity is of order $O((n+m)^9)$ for $A \in \mathbb{R}^{m \times n}$, which does not really help matters much.

The simplex algorithm as implemented in Algorithm 1 is not suited for large problems that appear in applications. There, often the number of variables is huge (several hundred thousands), but most are affected only by a comparably small number of inequalities. Then the matrix A consists mostly of zeros, that is, it is sparse. The problem in this situation is that the matrix A_B^{-1} that is computed during the simplex algorithm need not be sparse anymore. Then, instead of working with the simplex tableau, it is better to work directly with Theorem 2.3.4 and acquire the vectors $c_B^T A_B^{-1} A_{B'}$ and $(A_B^{-1} A_{B'})_{\cdot,j_0}$ that are needed for the update in a different manner. A typical method is to use an incomplete LU-factorisation (that is, the Gauß algorithm) and update the lower and upper matrices similarly as the simplex tableau in Algorithm 1.

Chapter 3

Integer Polyhedra

In the previous chapter, we have studied the solution of continuous linear optimisation problems. Before we now turn to the study of *discrete* linear programmes, we first discuss what the restriction to discrete solutions means for the set of feasible solutions.

3.1 Integer Polyhedra

Definition 3.1.1. A set $K \subset \mathbb{R}^n$ is called *convex*, if $\lambda x + (1-\lambda)x \in K$ whenever $x \in K$, $y \in K$, and $0 \le \lambda \le 1$.

Let $L \subset \mathbb{R}^n$ be any set. The *convex hull* of L, denoted as $\operatorname{conv}(L)$, is defined as the smallest convex set containing L. ■

Lemma 3.1.2. *Let $x^{(i)} \in \mathbb{R}^n$, $i = 1, \ldots, n$, be a finite family of points. Then the convex hull of the set $\{x^{(i)}\}_{1 \le i \le n}$ is a polytope. Moreover, the set of vertices of this polytope is contained in $\{x^{(i)}\}_{1 \le i \le n}$.*

Definition 3.1.3. A polyhedron P is called *rational*, if there exist a matrix $A \in \mathbb{Q}^{m \times n}$ and a vector $b \in \mathbb{Q}^m$ such that $P = \mathcal{P}(A, b)$.

In addition, if P is any polyhedron, then we denote by

$$P_I := \operatorname{conv}(P \cap \mathbb{Z}^n)$$

the *integer hull* of P. Moreover, we let $\mathcal{P}_I(A, b) := \big(\mathcal{P}(A, b)\big)_I$.

The polyhedron P is called *integral*, if $P = P_I$. ■

Lemma 3.1.4. *Assume that P is a polytope. Then so is P_I. Also, if P is a rational polyhedron, then P_I is a (rational) polyhedron, too.*

The next example shows that this result need not hold, if the polyhedron P is neither rational nor bounded.

Example 3.1.5. Consider the (non-rational) polyhedron

$$P = \left\{(x, y) \in \mathbb{R}^2 : x \ge 1,\ y \ge 0,\ -\sqrt{2}x + y \le 0\right\}.$$

Then the set P_I has infinitely many vertices and therefore is no polyhedron.

Consider in particular the minimization problem

$$\sqrt{2}x - y \to \min \qquad \text{subject to } (x, y) \in P_I \ .$$

This problem is bounded below by zero and is feasible, as it is easy to see that P_I is non-empty. Still, it admits no solution: Indeed, the infimum of the cost function is zero. Assume now that this infimum would be attained at some point $(x, y) \in P_I$. Then the infimum would also be attained at a vertex of P_I, and thus we could assume without loss of generality that $(x, y) \in \mathbb{Z}^2$. This, however, would imply that $\sqrt{2}$ were rational. Therefore, the integer problem has no optimal solution. ∎

Consider now a discrete linear programme of the form

$$c^T x \to \min \qquad \text{subject to } Ax \le b \text{ and } x \in \mathbb{Z}^n \ ,$$

where the matrix A and the vector b are rational. By definition of the integer hull of a polyhedron, this is equivalent to

$$c^T x \to \min \qquad \text{subject to } x \in \mathcal{P}_I(A, b) \cap \mathbb{Z}^n \ . \tag{3.1}$$

Because by assumption $\mathcal{P}(A, b)$ is a rational polyhedron, the set $\mathcal{P}_I(A, b)$ is a polyhedron, too. In particular, the relaxed problem

$$c^T x \to \min \qquad \text{subject to } x \in \mathcal{P}_I(A, b) \ , \tag{3.2}$$

where we have omitted the integrality condition, attains solutions at the vertices of $\mathcal{P}_I(A, b)$. These vertices, however, are elements of $\mathcal{P}_I(A, b) \cap \mathbb{Z}^n$ and therefore also solve the problem (3.1). This shows that (3.1) and (3.2) are equivalent in the sense that every vertex solution of (3.2) solves (3.1) (and conversely every solution of (3.1) also solves (3.1)). Because (3.2) is a continuous linear programme, we have thus demonstrated that integer linear programming is almost the same as continuous linear programming.

Still, when one considers again the problem (3.2) and compares it to the linear programmes studied in the previous chapter, one might notice a small, but important, difference. For continuous problems, we have always assumed that the bounds were given explicitely as some inequality $Ax \le b$. In (3.2), however, the restrictions are only given *implicitly* by the condition $x \in \mathcal{P}_I(A, b)$. This implicit definition of the bounds makes the problem much harder. Continuous linear programmes can be solved in polynomial time (though not really efficiently), integer problems, in contrast, in general probably cannot. Even more, already the sub-problem of deciding whether the problem is feasible is a hard problem. In order to make this statement more precise, we recall some definitions:

Definition 3.1.6.

- The class $\mathcal{P}$ consists of all decision problems that can be solved in polynomial time.
- The class $\mathcal{NP}$ consists of all decision problems for which a non-deterministic, polynomial time solution algorithm exists.

  Equivalently, a decision problem Π lies in $\mathcal{NP}$, if and only if there exists another decision problem Π' with the following properties:

  - The instances of Π' consist of concatenations $p\#c$ with $p \in \Pi$.
  - Whenever p is a *yes*-instance for Π and c is such that $p\#c \in \Pi'$, then $p\#c$ is a *yes*-instance for Π'.
  - For each *no*-instance $p \in \Pi$, there exists a string c such that $p\#c$ is a *no*-instance for Π'.
  - $\Pi' \in \mathcal{P}$.

  The strings c for which $p\#c \in \Pi'$ are called *certificates* for p.

- A problem Π is $\mathcal{NP}$-hard, if a polynomial algorithm for the solution of Π implies a polynomial algorithm for every problem in $\mathcal{NP}$.
- A decision problem Π is $\mathcal{NP}$-complete, if it is $\mathcal{NP}$-hard and lies in $\mathcal{NP}$.■

Remark 3.1.7.

- If $\Pi \in \mathcal{P}$, then every instance p can serve as its own certificate. Thus we have the inclusion $\mathcal{P} \subset \mathcal{NP}$.
- While the classes $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NP}$-complete only encompass *decision problems*, where the task is to decide whether some instance $p \in \Pi$ is a member of some subclass $L \subset \Pi$, the class $\mathcal{NP}$-hard also includes other types of problems like optimisation problems.

  If a *decision problem* is $\mathcal{NP}$-hard, but not $\mathcal{NP}$-complete, then there exists *no* polynomial time algorithm for its solution (else it would lie in $\mathcal{P}$ and therefore in $\mathcal{NP}$).
- The question whether $\mathcal{P} = \mathcal{NP}$ is still undecided; it is one of the most important open problems in complexity theory. ■

Proposition 3.1.8. *The decision problem whether a system of rational inequalities $Ax \leq b$ has an integer solution is $\mathcal{NP}$-complete.*

In particular, the previous proposition implies that the problem of linear integer programming is $\mathcal{NP}$-hard. Thus, most probably no efficient (in the sense of polynomial time) algorithms for the general solution of linear integer progammes exist. In the next sections we will therefore restrict ourselves to the study of the special situation where integer programming equals continuous programming, as the corresponding polyhedra coincide. These problems are solvable in polynomial time by the ellipsoid method, and also efficiently in practice by the simplex method. The general case will be then treated in the following chapters.

3.2 Total Unimodularity and Total Dual Integrality

Proposition 3.2.1. *Let P be a rational polyhedron. Then the following are equivalent:*

1. *P is an integral polyhedron.*
2. *Every face of P contains an integer vector.*
3. *Every minimal face of P contains an integer vector.*
4. *If $\xi \in \mathbb{R}^n$ is such that $\sup\{\xi^T x : x \in P\}$ is finite, then the maximum is attained at some integer vector. In other words, every supporting hyperplane of P contains an integer vector.*

Combining Propositions 2.1.4 and 3.2.1, we see that a rational polyhedron $P = \mathcal{P}(A, b)$ is integral, if every sub-system of the equation $Ax = b$ attains an integer solution. Thus it makes sense to study the sub-matrices of rational matrices $A \in \mathbb{Q}^{m\times n}$ in order to decide on the integrality of rational polyhedra.

Definition 3.2.2. A square matrix $A \in \mathbb{R}^{m\times m}$ is called *unimodular*, if its entries are integers and $\det A = \pm 1$. ■

Lemma 3.2.3. *The inverse of a unimodular matrix is also unimodular. For every unimodular matrix $U \in \mathbb{R}^{m\times m}$ the mappings $x \mapsto Ux$ and $x \mapsto xU$ are bijections on $\mathbb{Z}^m$. In particular, if U is unimodular and $x \in \mathbb{Z}^m$, then also $U^{-1}x \in \mathbb{Z}^m$.*

Definition 3.2.4. A matrix $A \in \mathbb{R}^{m\times n}$ is *totally unimodular*, if the determinant of every square sub-matrix is either 0 or ± 1. Equivalently, A is totally unimodular, if every invertible square sub-matrix is unimodular. ■

Remark 3.2.5. Note that the entries of totally unimodular matrices are all either 0 or ± 1, as they are precisely the 1×1 sub-matrices.

Moreover, in view of Remark 2.2.2 it is helpful to note that a matrix A is totally unimodular, if and only if either of the matrices (A, Id_m), $(A, -A, \mathrm{Id}_m)$, $(A, -A, \mathrm{Id}_m, -\mathrm{Id}_m)$, and A^T is unimodular. ■

Theorem 3.2.6. *Let $A \in \mathbb{R}^{m\times n}$ be an integer matrix. The matrix A is totally unimodular, if and only if the polyhedron*

$$P = \{x \in \mathbb{R}^n : Ax \le b,\ x \ge 0\}$$

is integral for all integer vectors $b \in \mathbb{Z}^m$.

Example 3.2.7. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. Denote, for given vertex $v \in V$ by $\delta^+(v) \subset E$ the edges starting in v, and by $\delta^-(v) \subset E$ the edges ending in v, that is,

$$\begin{aligned} \delta^+(v) &= \{e = (v, w) : w \in V,\ e \in E\}\,, \\ \delta^-(v) &= \{e = (w, v) : w \in V,\ e \in E\}\,. \end{aligned}$$

Denote by $A \in \mathbb{R}^{V\times E}$ the *incidence matrix*, defined by

$$a_{v,e} := \begin{cases} +1 & \text{if } e \in \delta^+(v)\,, \\ -1 & \text{if } e \in \delta^-(v)\,, \\ 0 & \text{else.} \end{cases}$$

Then A is totally unimodular. ■

Example 3.2.8. Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E. Denote, for $v \in V$, by $\delta(v) \subset E$ the edges containing the vertex v. That is, $\delta(v) = \big\{\{v, w\} : w \in V,\ \{v, w\} \in E\big\}$. Denote again by $A \in \mathbb{R}^{V\times E}$ the incidence matrix of G, which for an undirected graph is defined as

$$a_{v,e} := \begin{cases} +1 & \text{if } e \in \delta(v)\,, \\ 0 & \text{else.} \end{cases}$$

Then one can show that A is totally unimodular, if and only if G is bipartite.

Consider now again the assignment problem from Section 1.2.3, written as an optimisation problem on a graph. Then the incidence matrix of the graph defines precisely the necessary constraints: A matching between the vertices is described by the system of equations $Ae = 1$. Because the graph is bipartite, the matrix A is totally unimodular. As a consequence, the polytope defined by the equation $Ae = 1$ is integer. Thus, in theory, one could use the simplex algorithm for the solution of the assignment problem. In addition, the total unimodularity of A implies that the right hand side in the equation $Ae = 1$ can be replaced by any integer vector without losing the integrality of the corresponding polyhedron. Thus also different constraints than "one job per person" can be modelled.

Note, however, that more efficient algorithms for the solution of the assignment problems exist. Still, this result is interesting for theoretical reasons, and also because the assignment problem is a sub-problem of one formulation of the much more complicated traveling salesman problem. ■

While the total unimodularity of a matrix A implies that the polyhedron $\mathcal{P}(A, b)$ is integral for each integral right hand side $b \in \mathbb{Z}^m$, the assumption is too strong if we only want to have integrality for some given, *fixed* b. In this case, total dual integrality is the right concept to use.

Definition 3.2.9. Let $A \in \mathbb{Q}^{m\times n}$ and $b \in \mathbb{Q}^m$. A system $Ax \le b$ of linear inequalities is *totally dual integral*, if for every integer vector $c \in \mathbb{Z}^n$ for which

$$\sup\big\{b^T y : A^T y = c,\ y \ge 0\big\} < \infty$$

there exists an integer vector $y_0 \in \mathbb{Z}^m$ satisfying $A^T y_0 = c$, $y_0 \ge 0$, and

$$b^T y_0 = \max\big\{b^T y : A^T y = c,\ y \ge 0\big\}\ . \qquad ■$$

Remark 3.2.10. Consider a linear programme (the *primal* programme)

$$c^T x \to \min \qquad\qquad \text{subject to } Ax \le b\ .$$

Then we define its *dual* to be the linear programme

$$b^T y \to \max \qquad\qquad \text{subject to } A^T y = c \text{ and } y \ge 0\ .$$

If both the primal and the dual problem are feasible, then

$$\min\big\{c^T x : Ax \le b\big\} = \max\big\{b^T y : A^T y = c \text{ and } y \ge 0\big\}\ . \tag{3.3}$$

Even more, if x_0 and y_0 are feasible solutions of the primal and the dual problem, then the following statements are equivalent:

- The vectors x_0 and y_0 are both optimal solutions.
- $c^T x_0 = b^T y_0$.
- $y_0^T(b - Ax_0) = 0$.

Many optimisation algorithms (including the simplex algorithm) are based on this result relating the primal and the dual problem. ■

Using the formulation of the dual programme, the concept of total dual integrality can be brought in a more natural form: A system of linear inequalities $Ax \leq b$ is totally dual integral, if and only if for every integer cost vector $c \in \mathbb{Z}^n$ the corresponding dual problem has an integer solution. As a consequence, if both A and b are integral, then (3.3) implies that the optimal value of the primal problem is integral. This strongly suggests that the optimum is attained at an integer vector. Indeed, the following result holds:

Proposition 3.2.11. *Assume that the system of inequalities $Ax \leq b$ is totally dual integral and that $b \in \mathbb{Z}^m$ is an integer vector. Then the polyhedron $\mathcal{P}(A, b)$ is integral.*

Note that total dual integrality is a property of systems of inequalities, not of polyhedra. The next result shows, however, that every rational polyhedron can be described by a totally dual integral system of inequalities.

Proposition 3.2.12. *Let $P \subset \mathbb{R}^n$ be a rational polyhedron. Then there exist a matrix $A \in \mathbb{Q}^{m \times n}$ and a vector $b \in \mathbb{Q}^m$ such that $P = \mathcal{P}(A, b)$ and the system $Ax \leq b$ is totally dual integral.*

Corollary 3.2.13. *A rational polyhedron P is integral, if and only if there exists a matrix $A \in \mathbb{Z}^{m \times n}$ and a vector $b \in \mathbb{Z}^m$ such that $P = \mathcal{P}(A, b)$ and the system $Ax \leq b$ is totally dual integral.*

Chapter 4

Relaxations

4.1 Cutting Planes

The idea behind the method of cutting planes for the solution of an integer linear programme of the form

$$c^T x \to \min \qquad \text{subject to } Ax \le b\,, \text{ and } x \in \mathbb{Z}^n \tag{4.1}$$

is the following: First one considers the *LP-relaxation* of (4.1) defined as

$$c^T x \to \min \qquad \text{subject to } Ax \le b\,, \text{ and } x \in \mathbb{R}^n\,. \tag{4.2}$$

That is, one simply forgets for the moment about the integrality restriction. The relaxed problem (4.2) can be solved quite efficiently with the simplex algorithm. If the solution turns out to be integral, then we are done. This happens, for instance, if the system of inequalities $Ax \le b$ is totally dual integral, and A, b, and c are integral. Else the solution is a vertex x_0 (we may assume without loss of generality that A has full rank) that does not lie in the integer hull of the polyhedron $\mathcal{P}(A, b)$. Therefore there exists a hyperplane separating the vertex x_0 and the integer polyhedron $\mathcal{P}_I(A, b)$. This separation can be described by linear inequalities of the form

$$a_0^T x \le b_0 < a_0^T x_0 \qquad \text{for all } x \in \mathcal{P}_I(A, b)$$

for some $a_0 \in \mathbb{Z}^n$ and $b_0 \in \mathbb{Z}$. We can now add this new inequality $a_0^T x \le b_0$ to the system $Ax \le b$ and obtain the new linear programme

$$c^T x \to \min \qquad \text{subject to } Ax \le b\,,\ a_0^T x \le b_0\,, \text{ and } x \in \mathbb{Z}^n\,,$$

which we can solve. Because the old solution x_0 is not admissible for the new problem, we will obtain a different solution, which, hopefully, is now integral. Else one can repeat the process. Maybe unexpectedly, this method really works, provided the new inequalities one adds in each iteration are chosen in a suitable manner.

4.1.1 Gomory–Chvátal Truncation

Definition 4.1.1. Let P be a polyhedron. Define P' as the intersection of all integer hulls of rational, affine half-spaces containing P, that is,

$$P' := \bigcap_{H \in \mathcal{H}(P)} H_I \qquad \text{with } \mathcal{H}(P) := \bigl\{ H = \mathcal{P}(\xi, \delta) : \xi \in \mathbb{Q}^n,\ \delta \in \mathbb{Q},\ P \subset H \bigr\} .$$

Moreover let $P^{(0)} := P'$ and $P^{(i+1)} := (P^{(i)})'$. The set $P^{(i)}$ is called the *i-th Gomory–Chvátal truncation* of P. ■

One has the chain of inclusions

$$P \supset P^{(0)} \supset P^{(1)} \supset \cdots \supset P_I .$$

A different, but equivalent, way of defining the set P' is by considering only half-planes with integer coefficients. Indeed, it is easy to see that

$$P' = \bigl\{ x \in \mathbb{R}^n : \xi^T x \le \lfloor \delta \rfloor \text{ for all } \xi \in \mathbb{Z}^n,\ \delta \in \mathbb{Q} \text{ with } \xi^T y \le \delta \text{ for all } y \in P \bigr\} .$$

Here $\lfloor \delta \rfloor$ denotes the largest integer below δ. Even more, the following characterisation holds:

Proposition 4.1.2. *Let $P = \mathcal{P}(A, b)$ be a polyhedron with $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{R}^m$ rational. Then*

$$P' = \bigl\{ x \in \mathbb{R}^n : \xi^T A x \le \lfloor \xi^T b \rfloor \textit{ for all } \xi \in \mathbb{Z}^m \textit{ with } \xi \ge 0 \textit{ and } \xi^T A \in \mathbb{Z}^n \bigr\} .$$

Proposition 4.1.3. *Assume that either P is a rational polyhedron or P is a polytope. Then there exists a number $t \in \mathbb{N}$ such that $P^{(t)} = P_I$. That is, the truncation stops after a finite number of steps.*

In case the inequality $Ax \le b$ is totally dual integral, one can even compute the truncation in one step:

Proposition 4.1.4. *Assume that $P = \mathcal{P}(A, b)$ with $Ax \le b$ totally dual integral, $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. Then*

$$P' = \bigl\{ x \in \mathbb{R}^n : Ax \le \lfloor b \rfloor \bigr\} .$$

Thus, in principle, for solving an integer linear programme, it is sufficient to bring the inequalities in a totally dual integral form, truncate them, and then use any solution algorithm for the relaxed, truncated problem. The only problem of this approach is that computing a totally dual integral form of an arbitrary set of inequalities cannot be done efficiently.

4.1.2 Gomory's Algorithmic Approach

Assume that we are given an integer linear programme in canonical form, that is,

$$c^T x \to \min \qquad \text{subject to } Ax = b,\ x \ge 0,\ x \in \mathbb{Z}^n . \tag{4.3}$$

In addition, we assume that $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$ are integral.

Consider now the LP-relaxation of (4.3), given by

$$c^T x \to \min \qquad \text{subject to } Ax = b,\ x \geq 0,\ x \in \mathbb{R}^n\ . \tag{4.4}$$

If this problem is solved by the simplex method, then we obtain an optimal solution $\tilde{x}$ and a corresponding subset of indices $B \subset \{1, \dots, n\}$ with $|B| = m$ such that the matrix A_B is a feasible basis for (4.4). In particular,

$$\tilde{x}_i = \begin{cases} (A_B^{-1}b)_i & \text{if } i \in B\,, \\ 0 & \text{if } i \notin B\,. \end{cases}$$

Now let x be any feasible vector for (4.4). Then the equation $Ax = b$ holds, and therefore, with the notation $x_{B'} := (x_j)_{j \in B'}$,

$$x_i = (A_B^{-1}b)_i - (A_B^{-1}A_{B'}x_{B'})_i \qquad \text{for every } i \in B\ .$$

Consequently, if x is a feasible *integer* vector, then

$$(A_B^{-1}b)_i - (A_B^{-1}A_{B'}x_{B'})_i \in \mathbb{Z} \qquad \text{for every } i \in B$$

Now denote by $f\colon \mathbb{R} \to [0,1)$ the mapping $f(t) = t - \lfloor t \rfloor$. Then every feasible integer vector x satisfies

$$f((A_B^{-1}b)_i) - \sum_{j \in B'} f((A_B^{-1}A_{B'})_{i,j})x_j \in \mathbb{Z} \qquad \text{for every } i \in B$$

Because the mapping f takes values in $[0,1)$ and the feasibility of x implies that $x_j \geq 0$ for every $j \in B'$, it follows that every feasible integer vector x satisfies the system of inequalities

$$f((A_B^{-1}b)_i) - \sum_{j \in B'} f((A_B^{-1}A_{B'})_{i,j})x_j \leq 0\,,$$

or, equivalently,

$$\sum_{j \in B'} f((A_B^{-1}A_{B'})_{i,j})x_j \geq f((A_B^{-1}b)_i) \tag{4.5}$$

for every $i \in B$.

Now assume that the optimal solution $\tilde{x}$ of the relaxed problem is not integral. Denote by $I \subset B$ the set of indices i for which $\tilde{x}_i \notin \mathbb{Z}$. For all these indices we have $f(\tilde{x}_i) > 0$. Therefore, because $\tilde{x}_j = 0$ for $j \in B'$, it follows that the inequality (4.5) is violated for every $i \in I$, the left hand side being zero and the right hand side strictly positive. Thus we can add one of the inequalities (4.5) for some $i \in I$ to our system and repeat the process.

It is also possible to obtain again a system of *integral* inequalities. Some manipulation of (4.5) using the fact that $Ax = b$ for every feasible vector x shows that this inequality is equivalent to adding the integer inequality

$$x_i + \sum_{j \in B'} \lfloor (A_B^{-1}A_{B'})_{i,j} \rfloor x_j \leq \lfloor (A_B^{-1}b)_i \rfloor$$

for some $i \in I$, or, introducing a slack variable y,

$$y + x_i + \sum_{j \in B'} \lfloor (A_B^{-1}A_{B'})_{i,j} \rfloor\, x_j = \lfloor (A_B^{-1}b)_i \rfloor, \qquad y \geq 0\ .$$

It has been shown by Gomory that certain choices of the inequalities that are added in that manner indeed lead to a finite solution algorithm for the integer linear programme.

Remark 4.1.5. The method described above works only for integer linear programmes, not for mixed integer programmes. There exist, however, modifications that are also able to treat this more complicated case. ■

4.2 Lagrangean Relaxation

The idea of cutting planes was, to relax the integrality condition, but to retain the inequalities $Ax \le b$. The method described in this section, in contrast, keeps the integrality condition and relaxes some of the inequalities instead. More precisely, we omit some inequalities and instead reintroduce them in the cost functional as additional penalty terms. This makes sense, if the problem we obtain after the deletion of certain inequalities becomes much easier to solve (preferably by a combinatorial algorithm that does not rely on integer programming). We assume in the following that we are given an integer programme, but the method works also without any major modifications on mixed integer programmes.

Suppose we are given an integer linear programme of the form

$$c^T x \to \min \qquad \text{subject to } x \in \mathbb{Z}^n \text{ and } \begin{cases} A^{(1)}x \le b^{(1)}\,, \\ A^{(2)}x \le b^{(2)}\,, \end{cases} \tag{4.6}$$

where $A^{(1)} \in \mathbb{R}^{m_1 \times n}$, $b^{(1)} \in \mathbb{R}^{m_1}$, $A^{(2)} \in \mathbb{R}^{m_2 \times n}$, $b^{(2)} \in \mathbb{R}^{m_2}$.

Define now the *Lagrange functional* $L \colon \mathbb{R}^{m_1}_{\ge 0} \to \mathbb{R}$ as

$$L(\lambda) := \inf\bigl\{c^T x - \lambda^T(b^{(1)} - A^{(1)}x) : x \in \mathcal{P}(A^{(2)}, b^{(2)}) \cap \mathbb{Z}^n\bigr\}\,.$$

That is, for the minimisation involved in the definition of L we forget about the first constraint in (4.6), but instead increase the cost by the additional penalty $\lambda^T(b^{(1)} - A^{(1)}x)$ if the condition $A^{(1)}x \le b^{(1)}$ is violated.

Now note that the solution $\tilde{x}$ of (4.6) is also admissible for the minimisation problem

$$c^T x - \lambda^T(b^{(1)} - A^{(1)}x) \to \min \qquad \text{subject to } A^{(2)}x \le b^{(2)},\ x \in \mathbb{Z}^n\,, \tag{4.7}$$

and, by the admissibility of $\tilde{x}$ for (4.6), $b^{(1)} - A^{(1)}x \le 0$. Because $\lambda \ge 0$, it follows that

$$\begin{aligned} L(\lambda) &\le c^T\tilde{x} - \lambda^T(b^{(1)} - A\tilde{x}) \\ &\le c^T\tilde{x} = \min\bigl\{c^T x : x \in \mathbb{Z}^n \cap \mathcal{P}(A^{(1)}, b^{(1)}) \cap \mathcal{P}(A^{(2)}, b^{(2)})\bigr\}\,. \end{aligned}$$

This argumentation being valid for every $\lambda \ge 0$, it follows that also $\max_{\lambda \ge 0} L(\lambda)$ is a lower bound for the minimal value of the original minimisation problem (4.6). More precisely, it is possible to show that the following relation holds:

Proposition 4.2.1. *Assume that $\mathcal{P}(A^{(2)}, b^{(2)})$ is a polytope. Assume in addition that $\mathcal{P}(A^{(2)}, b^{(2)}) \cap \mathbb{Z}^n$ is non-empty. Then*

$$\max\bigl\{L(\lambda) : \lambda \in \mathbb{R}^{m_1},\ \lambda \geq 0\bigr\} = \min\bigl\{c^T x : x \in \mathcal{P}(A^{(1)}, b^{(1)}) \cap \mathcal{P}_I(A^{(2)}, b^{(2)})\bigr\}\ .$$

The last result shows that the maximal value of the Lagrange functional equals the minimal value of a relaxation of the problem (4.6). In the special case, where the polyhedron $\mathcal{P}(A^{(1)}, b^{(1)})$ is integral, that is, if

$$\mathcal{P}(A^{(1)}, b^{(1)}) = \mathcal{P}_I(A^{(1)}, b^{(1)})\,,$$

it follows that $\max_{\lambda \geq 0} L(\lambda)$ is precisely the same as the optimal value of the original problem.

Remark 4.2.2. Let λ^* be the maximiser of the Lagrange functional and let $x^* \in \mathbb{Z}^n$ be a solution of (4.7) with λ replaced by λ^*. Then by definition $A^{(2)}x^* \leq b^{(2)}$. If, in addition, the inequality $A^{(1)}x^* \leq b^{(1)}$ is satisfied, then x^* is also a solution of (4.6). In general, however, this inequality will not be satisfied, and therefore x^* will not be admissible for the original problem. Still, in some problems it is possible to modify x^* in such a way that also the first inequality is satisfied, but the value of the cost functional does not change too much. More important, however, are applications where one mainly needs an estimate for the optimal value of the cost functional. For instance, this is the case for branch-and-bound methods to be discussed in Section 6.1. ■

In order to make use of Proposition 4.2.1, it is necessary to find an optimal Lagrange parameter λ. Because of the special structure of the Langrange functional, this can be done with a gradient based approach, which is described in the next proposition.

Proposition 4.2.3. *Let $\lambda^0 \in \mathbb{R}^{m_1}_{\geq 0}$ be arbitrary, and let $(\mu^{(k)})_{k\in\mathbb{N}}$ any sequence of positive numbers such that $\lim_{k\to\infty} \mu^{(k)} = 0$ and $\sum_{k\in\mathbb{N}} \mu^{(k)} = +\infty$. Define inductively*

$$x^{(k)} := \arg\min_x \bigl\{c^T x - (\lambda^{(k)})^T (b^{(1)} - A^{(1)}x) : x \in \mathcal{P}_I(A^{(2)}, b^{(2)})\bigr\}$$

and

$$\lambda^{(k+1)} = \lambda^{(k)} + \mu^{(k)}(A^{(1)}x^{(k)} - b^{(1)})\ .$$

Then the sequence $(\lambda^{(k)})_{k\in\mathbb{N}}$ converges to a maximiser λ^* of L.

Remark 4.2.4. The method described in Proposition 4.2.3 is a special instance of a *sub-gradient method.* More details will be found in the lecture “Continuous Optimisation.” ■

Remark 4.2.5. The condition in Proposition 4.2.3 that the sequence $\mu^{(k)}$ converges to zero is not really required. In fact, one only needs that $\mu^{(k)}\|A^{(1)}\| < 1$ for k sufficiently large, else the algorithm will diverge. ■

Remark 4.2.6. The algorithm described in Proposition 4.2.3 requires that the Lagrange relaxation is solved multiple times with different values of the Lagrange parameter λ. This is only feasible, if the relaxed problem can be solved much more efficiently than the original problem. This can happen if the relaxed problem is much smaller than the original one; in this case, however, the estimate of the value of the original problem might not be satisfactory, as a large part of the inequalities have been relaxed. More important is the situation, where the relaxed problem can be solved by an efficient combinatorial algorithm. For an example, see Section 6.1.1. ■

Chapter 5

Heuristics

5.1 The Greedy Method

Typical binary optimisation problems can be formulated, or naturally appear, as optimisation problems on certain classes of subsets of some given finite set. That is, we are given a finite set E and a class $\mathcal{F}$ of subsets of E. In addition, we have a cost function $c\colon \mathcal{F} \to \mathbb{R}$. The goal is to find $F \in \mathcal{F}$ such that $c(F)$ is minimal (or maximal). Moreover, in many situations the function c is *modular*, that is, whenever X, $Y \in \mathcal{F}$ are disjoint and $X \cup Y \in \mathcal{F}$, we have $c(X \cup Y) = c(X) + c(Y)$.

Definition 5.1.1. Let E be a set and $\mathcal{F}$ a class of subsets of E. The pair $(E, \mathcal{F})$ is an *independence system* if the following two conditions hold:

- $\emptyset \in \mathcal{F}$.
- If $Y \in \mathcal{F}$ and $X \subset Y$, then also $X \in \mathcal{F}$.

The sets $F \in \mathcal{F}$ are called *independent*, all the others are called *dependent*. If $X \subset E$, the maximal independent sets in X are called *bases* of X (that is, a set F is a basis of X, if $F \in \mathcal{F}$ and there exists no $G \in \mathcal{F}$ with $F \subsetneq G \subset X$). ■

The following two problems are of main interest for independence systems:

- Maximise the (modular) cost function c over the independence system $(E, \mathcal{F})$ (maximisation problem over an independence system).
- Minimise the (modular) cost function c over the set of all bases of E with respect to the independence system $(E, \mathcal{F})$ (minimisation problem over a basic system).

1. Let $G = (V, E)$ be a connected (undirected) graph with edge set E and consider the family $\mathcal{F}$ of all forests in G written as subsets of E. Then it is easy to see that the pair $(E, \mathcal{F})$ is an independence system, as every subgraph of a forest is itself a forest. Thus, the maximisation problem over $(E, \mathcal{F})$ is the task of finding a maximum weight forest in G.

   If the graph G is connected, the bases of E with respect to $(E, \mathcal{F})$ are precisely the spanning trees of G. The minimisation problem over the basic system therefore is the task of finding a minimum weight spanning tree.

2. *Traveling Salesman Problem (TSP):* Find a minimal Hamiltonian circuit in a given complete (undirected) graph $G = (V, E)$ with respect to a family of weights c.

   Here one can define the independence system $(E, \mathcal{F})$ as the family of all subsets of Hamiltonian circuits; obviously, the bases of the set E are then precisely the Hamiltonian circuits in G. The TSP is therefore the minimisation problem over the basic system with respect to $(E, \mathcal{F})$.

3. *Knapsack problem:* Given non-negative costs c_i, $1 \leq i \leq n$, weights a_i, $1 \leq i \leq n$, and some upper bound $b > 0$, find a set $S \subset \{1, \dots, n\}$ such that $\sum_{i \in S} a_i \leq b$ and $\sum_{i \in S} c_i$ is maximal. This is obviously the maximisation problem with respect to c over the independence system of all subsets S of $\{1, \dots, n\}$ of total weight $\sum_{i \in S} a_i$ at most b.

The greedy algorithm is a heuristic method for finding an approximation of a solution of either the maximisation or the minimisation problem. The underlying idea for the maximisation problem is, starting with the empty set as a candidate, to successively enlarge it by adding the element of the largest weight. Alternatively, one can start with the whole set as a candidate of the solution and then remove successively those elements of the smallest weight. In addition when adding elements, one has to guarantee in each step that the candidate remains independent. Similarly, if one removes elements, one has to stop as soon as one arrives at a basis.

Thus we end up with the following two algorithms for the maximisation problem—the algorithms for the minimisation problem can be written similarly; only the order of the elements has to be reversed:

```
Data: An independence system $(E, \mathcal{F})$ and a weight function $w$ on $E$;
Result: A set $F \subset E$;
Initialisation: Set $F := \emptyset$;
Order the elements of $E$ in such a way that $w(e_1) \geq w(e_2) \geq \ldots \geq w(e_n)$;
foreach $i = 1, \dots, n$ do
    if $F \cup \{e_i\} \in \mathcal{F}$ then
        $F \leftarrow F \cup \{e_i\}$;
    end
end
```

Algorithm 2: Best–in–greedy algorithm for the maximisation problem over an independence system

Typically, the ordering of the elements is determined by their cost. That is, $w(e_i) = c(e_i)$ for all i. In some applications, however, it is advisable to employ a weight different from the cost. This is for instance the case for the knapsack problem, where one obtains better results when one uses for the ordering the *relative cost* $w(e_i) := c(e_i)/a(e_i)$.

Now the question arises, whether the greedy algorithm is capable of finding the optimal solution. In most cases it is not. For a certain class of independence systems, however, it really does find the optimum.

```
Data: An independence system $(E, \mathcal{F})$ and a weight function $w$ on $E$;
Result: A basis $F \subset E$;
Initialisation: Set $F := E$;
Order the elements of $E$ in such a way that $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$;
foreach $i = 1, \ldots, n$ do
    if $F \setminus \{e_i\}$ either is a basis or contains a basis then
        $F \leftarrow F \setminus \{e_i\}$;
    end
end
```

Algorithm 3: Worst–out–greedy algorithm for the maximisation problem over a basis

Definition 5.1.2. An independence system $(E, \mathcal{F})$ is a *matroid* if, whenever X, $Y \in \mathcal{F}$ with $|X| > |Y|$, there exists $x \in X \setminus Y$ with $Y \cup \{x\} \in \mathcal{F}$.

Equivalently, $(E, \mathcal{F})$ is a matroid if, whenever $F \subset E$ and B, B' are bases of F, then $|B| = |B'|$. ■

Remark 5.1.3. Note that the definition of a matroid requires that for each subset F of E its bases are all of the same size; it is not sufficient that all bases of the whole set are of equal size. Thus the independence system of all subsets of Hamiltonian circuits is no matroid (if the underlying complete graph has at most four vertices), although every Hamiltonian circuit has the same number of edges. ■

Proposition 5.1.4. *An independence system $(E, \mathcal{F})$ is a matroid, if and only if for every modular cost function $c\colon E \to \mathbb{R}$ the best–in–greedy algorithm for the maximisation problem with weights equal to c finds an optimal solution.*

Similar results hold for the worst–out–greedy algorithm and also for the application of the two algorithms to the minimisation problem.

In case the independence system $(E, \mathcal{F})$ is no matroid, the greedy algorithm can perform arbitrarily badly. Then, also with different orderings, it is most of the time not possible to find an optimal solution.

Example 5.1.5 (Minimum spanning trees). Consider the problem of finding a minimum weight spanning tree in a graph $G = (V, E)$ with respect to a cost function $c\colon E \to \mathbb{R}$. In order to apply a greedy algorithm, we first order the edges with increasing weight. Then, starting with the empty set, we successively add an edge of smallest weight, unless this would result in a graph containing a circuit (and thus not being a forest anymore). In principle, we therefore have to test for circuits whenever we want to add an edge. Such a test can be performed in $O(n)$ time with n being the number of vertices (noting that a tree can contain at most $n - 1$ edges). As at most m tests have to be performed (m being the number of edges), and the sorting of the edges takes $O(m \log m)$ time, the computation time for the whole algorithm amounts to $O(mn)$. One can do better, however, if one keeps track of the connected components of the candidate tree during the iteration, because a circuit is formed if and only if the vertices of a candidate edge belong to the same connected component of the candidate tree.

With this modification, the whole algorithm can be implemented in $O(m \log n)$ time with m being the number of edges and n the number of vertices.

Because the family of forests in a graph is a matroid, this strategy is guaranteed to find an optimal solution. ■

5.1.1 Greedoids

The greedy algorithm can also be formulated for a different structure, where the main axiom defining independence systems is replaced by the axiom defining matroids.

Definition 5.1.6. Let E be a finite set and $\mathcal{F}$ a family of subsets of E. The pair $(E, \mathcal{F})$ is a *greedoid*, if the following two conditions hold:

- $\emptyset \in \mathcal{F}$.
- Whenever X, $Y \in \mathcal{F}$ with $|X| > |Y|$, there exists $x \in X \setminus Y$ such that $Y \cup \{x\} \in \mathcal{F}$. ■

Because of the second axiom, every independent set can be constructed within $\mathcal{F}$ from the empty set by successively adding single elements. Thus it is again possible to construct candidates for the optimal solution in the maximisation (minimisation) problem by adding elements of largest (smallest) weight.

```
Data: A greedoid $(E, \mathcal{F})$ and a weight function $w$ on $E$;
Result: A set $F \in \mathcal{F}$;
Initialisation: Set $F := \emptyset$;
Set $K := \{e \in E : \{e\} \in \mathcal{F}\}$;
while $K \neq \emptyset$ do
    Find $e \in K$ such that $w(e)$ is maximal;
    Set $F \leftarrow F \cup \{e\}$;
    Set $K := \{e \in E \setminus F : F \cup \{e\} \in \mathcal{F}\}$;
end
```

Algorithm 4: Best–in–greedy algorithm for the maximisation problem over a greedoid

Example 5.1.7 (Minimum spanning trees). We consider again the problem of finding a minimum weight spanning tree in a graph $G = (V, E)$ with respect to a cost function $c \colon E \to \mathbb{R}$. In contrast to the method described above, where we have considered the independence system of all forests in G, we now choose some vertex $v \in V$ and consider the greedoid of all trees rooted in v. Then the best–in–algorithms reads as follows: We start with the tree $T = (\{v\}, \emptyset)$. As long as the vertex set of T is not equal to V, we find the shortest edge leaving T and add it to T.

Again, this strategy is guaranteed to find an optimal solution. While the previous method can be implemented in $O(m \log n)$ time, this method can be implemented in $O(n^2)$ time (m – the number of edges; n – the number of vertices). For dense graphs, where $m \sim n^2$, this is preferable. ■

5.2 Local Search

In general, greedy and other heuristic methods will not lead to optimal solutions, but only to reasonably good candidates. Thus one might want to post-process the output of the heuristic method in order to obtain a result that is closer to the actual minimum. If the candidate solution is sufficiently good, it is reasonable to search for the true optimum only in a neighbourhood of this candidate—whatever *neighbourhood* means in this case.

The basic assumption of *local search* algorithms is that a meaningful notion of *closedness* is available. In the setting of optimisation problems on a class $\mathcal{F}$ of subsets of a given set E, this means that we can associate with each admissible set $X \in \mathcal{F}$ a *neighbourhood* $\mathcal{N}(X) \subset \mathcal{F}$. Starting with the output F of a heuristic method of choice, we then replace F by the minimum of the weight function w on the neighbourhood $\mathcal{N}(F)$ and repeat this process, until we arrive at a local optimum, that is, if F itself is the minimum of w on $\mathcal{N}(F)$. This method may work well, if the neighbourhoods $\mathcal{N}(F)$ are on the one hand small enough as to allow for a fast minimisation of w, and large enough for the method not to become trapped too fast in a local minimum.

```
Data: A family $\mathcal{F}$, a weight function $w$ on $\mathcal{F}$, for each set $X \in \mathcal{F}$ a
      family $\mathcal{N}(X) \subset \mathcal{F}$, an initial guess $X \in \mathcal{F}$;
Result: A set $F \in \mathcal{F}$;
Initialisation: Set $F := X$;
Set $\mathcal{K} := \{G \in \mathcal{N}(F) : w(G) < w(F)\}$;
while $\mathcal{K} \neq \emptyset$ do
    Choose $G \in \mathcal{K}$ of minimal weight;
    Set $F \leftarrow G$;
    Set $\mathcal{K} \leftarrow \{G \in \mathcal{N}(F) : w(G) < w(F)\}$;
end
```

Algorithm 5: Local search

Alternatively, it is also possible to choose *any* element $G \in \mathcal{N}(F)$ satisfying $w(G) < w(F)$. In practice this means that one scans the neighbourhood until one finds the first element of lower weight than F. Thus each iteration can be expected to be considerably faster than in the other approach, but, at the same time, the gain in each step will be smaller, and therefore a larger number of steps is needed to reach a local minimum. In addition, it is advisable to prescribe a maximal number of iterations or scans.

Example 5.2.1. Consider the traveling salesman problem, where $\mathcal{F}$ is the family of all Hamiltonian circuits in a given complete graph G. Then one can define, for a given number $k \geq 2$, the k-neighbourhood $\mathcal{N}_k(T)$ of a tour T as the family of those tours T' that differ from T by at most k edges.

$k = 2$: First note that two Hamiltonian circuits cannot differ by precisely one edge. Thus we have only have to consider exchanges of two edges. In order to find all possible exchanges, it makes sense to scan all pairs of edges to be replaced and then, for each of those pairs, consider all possible replacements that again lead to a Hamiltonian circuit. It is easy to see

```
Data: A family $\mathcal{F}$, a weight function $w$ on $\mathcal{F}$, for each set $X \in \mathcal{F}$ a
      family $\mathcal{N}(X) \subset \mathcal{F}$, an initial guess $X \in \mathcal{F}$;
Result: A set $F \in \mathcal{F}$;
Initialisation: Set $F := X$;
Set $\mathcal{K} := \mathcal{N}(F)$;
while $\mathcal{K} \neq \emptyset$ do
    Choose $G \in \mathcal{K}$;
    if $w(G) < w(F)$ then
        Set $F \leftarrow G$;
        Set $\mathcal{K} \leftarrow \mathcal{N}(G)$;
    else
        Set $\mathcal{K} \leftarrow \mathcal{K} \setminus \{G\}$;
    end
end
```

Algorithm 6: Local search; alternative

that all pairs of non-consecutive edges can be replaced in a unique manner, while for consecutive edges no replacement exists. Thus we simply have to enumerate all pairs of non-consecutive edges; the total number of computations is therefore of order n^2.

$k = 3$: In this case, we have first to scan all pairs of edges like in the case $k = 2$, and then all triples. Again, if the three edges to be replaced are consecutive, then no replacement exists. Also, if only two are consecutive, they can be replaced in a unique manner, while, in the general case, four replacements exist. The total number of computations is of order n^3.

As these examples indicate, the size of the neighbourhood $\mathcal{N}_k(T)$ is of order n^k. Thus, local search can only be applied for rather small k. In practice, a local search with $k = 3$ often yields good results, especially for graphs satisfying the triangle inequality. If stuck in a local minimum, though, it can be advisable to temporarily increase the neighbourhood.

Note, however, that, assuming $\mathcal{P} \neq \mathcal{NP}$, it can be shown that for every $k \in \mathbb{N}$ there exist examples of weighted graphs, where a local search with k-neighbourhoods does not yield an optimal result. Even more, it is also impossible to guarantee the value of the local optimum to lie within a prescribed percentage of the true value. ■

5.2.1 Tabu Search

One major problem of local search is that the iteration usually gets stuck in a local minimum. One way of escaping these minima is the enlargement of the neighbourhood. This enlargement, however, may vastly increase computation times while often not being sufficient for obtaining better results.

A different approach is to allow the solution also to increase when stuck in a minimum. For instance, it is possible to choose as an update the minimal element in $\mathcal{N}(F) \setminus \{F\}$, even if the weight of the element is larger than that of F. The problem with this idea is that, usually, one step will not be sufficient for leaving the local minimum. Then it is likely that in the next step we will simply return to the previous iterate. In order to avoid this *cycling*, it is possible to declare the elements already visited during the iteration *tabu* for the update. Or, in order to ensure that we do not even get near previous iterates, we might as well declare all the neighbourhoods of all already visited elements tabu. Since this is very memory consuming and also the computation time depends heavily on the size of the tabu list (in practice, we will have to decide separately for each possible update whether it is tabu or not), one typically only stores a limited number of previous updates in the tabu list.

```
Data: A family $\mathcal{F}$, a weight function $w$ on $\mathcal{F}$, for each set $X \in \mathcal{F}$ a
        family $\mathcal{N}(X) \subset \mathcal{F}$, an initial guess $X \in \mathcal{F}$;
Result: A set $F \in \mathcal{F}$;
Initialisation: Set $F := X$;
Set $G := X$;
Set $\mathcal{K} := \mathcal{N}(G)$;
Set $\mathcal{T} := \emptyset$;
while a stopping criterion is not yet satisfied do
    Choose $H \in \mathcal{K} \setminus \mathcal{T}$ of minimal weight;
    Set $G \leftarrow H$;
    if $w(G) < w(F)$ then
        Set $F \leftarrow G$;
    end
    Set $\mathcal{K} \leftarrow \mathcal{N}(G)$;
    Update the tabu list $\mathcal{T}$;
end
```

Algorithm 7: Tabu search

Apart from the choice of the neighbourhoods and the initial guess, there are two additional parameters that determine the performance of the method: the tabu list and the stopping criterion. If the tabu list is too large, then the algorithm can be very time and memory consuming. On the other hand, if the tabu list is too small, then there is the possibility of cycling. For the stopping criterion, the simplest choice is to prescribe a maximal number of iterations. Another common choice is to stop the iteration after a prescribed number of iterations without decreasing the objective function. All parameters depend heavily on the problem one wants to solve and, usually, have to be found by performing a large number of test runs.

5.2.2 Simulated Annealing

Similarly as tabu search, also *simulated annealing* allows the weight of the iterates to increase temporarily. The difference is that we do not scan the whole neighbourhood of an iterate, but, as in the second approach to local search, we choose the first candidate of smaller weight than the present iterate. In addition, we also allow updates of larger weight, but only with a certain probability depending on the weight difference and a *temperature*, which decreases during the iteration, until some *freezing temperature* is reached.

The idea behind this method comes from theoretical physics, where similar models are used for describing controlled cooling with phase transitions and cristallisation phenomena. Also the notions of temperature and freezing are due to this motivation.

```
Data: A family $\mathcal{F}$, a weight function $w$ on $\mathcal{F}$, for each set $X \in \mathcal{F}$ a family
      $\mathcal{N}(X) \subset \mathcal{F}$, an initial guess $X \in \mathcal{F}$, an initial temperature $T > 0$;
Result: A set $F \in \mathcal{F}$;
Initialisation: Set $F := X$;
Set $G := X$;
while freezing temperature not yet reached do
    Choose a random $H \in \mathcal{N}(G)$;
    if $w(H) < w(G)$ then
        Set $G \leftarrow H$;
        if $w(H) < w(F)$ then
            Set $F \leftarrow H$;
        end
    else
        choose a random number $\alpha \in [0, 1]$;
        if $\alpha < \exp\big((w(F) - w(H))/T\big)$ then
            Set $G \leftarrow H$;
        end
    end
    Update the temperature $T$;
end
```

Algorithm 8: Simulated annealing

It can be shown that, for suitable updates of the temperature, simulated annealing will (almost surely) indeed find the global optimum; there is, however, no bound on the running time. Indeed, in practical applications one has to run the algorithm for a very long time and decrease the temperature very slowly in order to obtain reasonable results. Also, a good choice of the decrease of the temperature has usually to be found in a long series of test runs.

Chapter 6

Exact Methods

6.1 Branch-and-Bound

Assume that we are given a *binary* linear programme

$$c^T x \to \min \qquad \text{subject to } Ax \le b \text{ and } x \in \{0,1\}^n \,. \tag{6.1}$$

Then it is, in principle, possible to solve the problem by considering all admissible values of x; testing whether the vector x is feasible; if it is, evaluating the objective functional at x; and in the end, selecting from all those values the minimal one.

The problem being binary, the enumeration of the admissible vectors x can be performed in a quite structured way, giving rise to the following recursion: Select some index $1 \le i^0 \le n$ and define two smaller problems by simply fixing the value of x_{i^0} to the two different possibilities. That is, consider the two problems

$$\begin{aligned}
&c^T x \to \min \qquad \text{subject to } Ax \le b \text{ and } x \in \{0,1\}^n \,, \text{ and } x_{i^0} = 0 \,,\\
&c^T x \to \min \qquad \text{subject to } Ax \le b \text{ and } x \in \{0,1\}^n \,, \text{ and } x_{i^0} = 1 \,.
\end{aligned}$$

Equivalently, these problems can be written as the two smaller binary programmes

$$\sum_{i \ne i_0} c_i x_i \to \min \qquad \text{subject to} \qquad \begin{aligned} &\sum_{i \ne i_0} A_{i,j} x_i \le b_j \text{ for all } j \,,\\ &x_i \in \{0,1\} \text{ for all } i \ne i_0 \,, \end{aligned} \tag{6.2}$$

and

$$\sum_{i \ne i^0} c_i x_i + c_{i^0} \to \min \qquad \text{subject to} \qquad \begin{aligned} &\sum_{i \ne i^0} A_{i,j} x_i \le b_j - A_{i^0,j} \text{ for all } j \,,\\ &x_i \in \{0,1\} \text{ for all } i \ne i_0 \,. \end{aligned} \tag{6.3}$$

Test whether the two programmes (6.2) and (6.3) are feasible. If neither is, then also the original programme (6.1) has an empty domain. If only one problem turns out to be feasible, then the solution of that programme will also be the solution of the original one. Finally, if both problems are feasible, then compute the solutions $x^{(0)}$ (with $x^{(0)}_{i0} = 0$) and $x^{(1)}$ (with $x^{(1)}_{i0} = 1$) and the corresponding values $v^{(0)} = c^T x^{(0)}$ and $v^{(1)} = c^T x^{(1)}$. If $v^{(0)} \le v^{(1)}$, then $x^{(0)}$ solves the original problem; else we obtain the solution $x^{(1)}$.

Each of the sub-problems (6.2) and (6.3) can be approached by the same method. If we decide to do so, then we indeed arrive at an enumeration method, consecutively, or in parallel, computing all possible values of the problem. More formally, this method can be described as follows:

```
Data: Binary linear programme:
      $P_0 = (c^T x \to \min$ subject to $Ax \le b,\ x \in \{0,1\}^n)$
Result: Either a solution $x$ or the knowledge that the linear programme
        is not feasible;
Initialisation: Define the list of problems $\mathcal{L} := \{P_0\}$, set $V := +\infty$,
                $z := \emptyset$.
while $\mathcal{L} \ne \emptyset$ do
    Choose any problem $P \in \mathcal{L}$ and set $\mathcal{L} \leftarrow \mathcal{L} \setminus \{P\}$;
    Find a partition $P = P_1 \dot\cup P_2$;
    foreach $i = 1, 2$ do
        if you can decide easily that $P_i$ is not feasible then
            do nothing;
        else if you can easily minimise $P_i$ then
            compute a minimiser $x$;
            if $c^T x < V$ then
                set $V \leftarrow c^T x$ and $z \leftarrow x$;
            end
        else
            set $\mathcal{L} \leftarrow \mathcal{L} \cup \{P_i\}$;
        end
    end
end
If $z = \emptyset$, then the linear programme $P_0$ is not feasible, else $x$ is a solution
and $V$ its value.
```

Algorithm 9: Enumeration method

Obviously, this approach will, in general, be not very efficient. A simple observation, however, can significantly reduce the computation time and make this approach efficient. Assume to that end that we already know some upper bound U for the value of the original problem (6.1). Such a value can, for instance, be obtained by finding *any* feasible vector $\hat{x}$ and setting $U := c^T \hat{x}$. Another possibility is to simply set $U := +\infty$. Then, if we can show that the minimal *value* of some sub-problem P_i is for sure larger than V, we can simply discard the problem. That is, we need not perform a further branching of the problem P_i, if we can be sure that *every* feasible solution x of P_i satisfies $c^T x > V$.

```
Data: Binary linear programme:
        $P_0 = (c^T x \to \min$ subject to $Ax \leq b,\ x \in \{0,1\})$
Input: Any upper bound $V$ for the value of $P_0$; if no upper bound is
        given, set $V := +\infty$;
Result: Either a solution $x$ or the knowledge that the linear programme
        is not feasible;
Initialisation: Define the list of problems $\mathcal{L} := \{P_0\}$, set $z := \emptyset$.
while $\mathcal{L} \neq \emptyset$ do
    Choose any problem $P \in \mathcal{L}$ and set $\mathcal{L} \leftarrow \mathcal{L} \setminus \{P\}$;
    Find a partition $P = P_1 \dot\cup P_2$ (branching);
    foreach $i = 1, 2$ do
        if you can decide easily that $P_i$ is not feasible then
            do nothing;
        else if you can easily minimise $P_i$ then
            compute a minimiser $x$;
            if $c^T x < V$ then
                set $V \leftarrow c^T x$ and $z \leftarrow x$;
            end
        else
            compute a lower bound $L$ for the cost of every feasible solution
            of the problem $P_i$ (bounding);
            if $L > V$ then
                do nothing;
            else
                set $\mathcal{L} \leftarrow \mathcal{L} \cup \{P_i\}$;
            end
        end
    end
end
If $z = \emptyset$, then the linear programme $P_0$ is not feasible, else $x$ is a solution
and $V$ its value.
```

Algorithm 10: Brand-and-Bound method

With this modification, we discard all the branches of the search tree that cannot contain the true solution. This can speed up the algorithm a lot, provided the following two conditions are satisfied: First, the computation of a lower bound must not be too costly, else the computation time will increase rather than decrease. Second, the lower bound should be as close as possible to the actual minimum of P. This will strongly increase the probability that we can actually discard a branch.

The easiest way for implementing the bounding step is to simply solve the LP-relaxation of the considered problem. That is, if the problem P_i reads as

$$\tilde{c}^T \tilde{x} \to \min \qquad \text{subject to } \tilde{A}\tilde{x} \leq \tilde{b} \text{ and } \tilde{x} \in \{0,1\}^k ,$$

we consider instead the relaxation

$$\tilde{c}^T \tilde{x} \to \min \qquad \text{subject to } \tilde{A}\tilde{x} \leq \tilde{b} \text{ and } 0 \leq \tilde{x} \leq 1 . \tag{6.4}$$

The latter problem can for instance be solved with the simplex algorithm. Because the solution is sought for on the polytope $\mathcal{P}(\tilde{A}, \tilde{b})$, which is larger than the polytope of the binary problem, the value of the relaxed problem is smaller or equal the value of the boundary problem. Thus we can use it as the lower bound L in the bounding step. In addition, if the relaxed problem turns out to be infeasible, then so is the binary problem. Also, if the solution of the relaxation is binary, then it already solves the binary problem. Thus, in fact, the relaxation (6.4) can be used for all the tasks necessary in the bounding step.

In order to obtain better lower bounds L, it makes sense to use some Lagrangean relaxation of the sub-problem P_i instead of the simple LP-relaxation. Of course, this is only possible, if the problem we are trying to solve has some nice structure, where we can easily identify sub-problems that can be solved with combinatorial algorithms. One such problem is the traveling salesman problem, where it is possible to seperate the constraints in such a way that the problem to be solved in the Lagrangean relaxation is that of finding some minimum weight spanning tree.

6.1.1 Application to the Traveling Salesman Problem

The following considerations largely follow the presentation in *Korte and Vygen, 2000*.

Consider again the Traveling Salesman Problem (TSP): Given a complete (undirected) graph G with n vertices and costs $c_{i,j} > 0$, $1 \le i < j \le n$, attached to the edge between i and j, find a Hamiltonian circuit C in G such that $c(C)$ is minimal. The TSP can be equivalently formulated as the binary linear programme (cf. Section 1.2.4)

$$\sum_{i,j} x_{i,j} c_{i,j} \to \min$$

subject to the constraints

$$\begin{aligned} 0 \le x_{i,j} &\le 1 && \text{for all } 1 \le i < j \le n\,, \\ \sum_{i=1}^{k-1} x_{k,i} + \sum_{i=k+1}^{n} x_{i,k} &= 2 && \text{for all } 1 \le k \le n\,, \\ \sum_{i<j\in I} &\le |I| - 1 && \text{for all } \emptyset \ne I \subsetneq \{1,\dots,n\}\,, \end{aligned} \tag{6.5}$$

and the additional condition

$$x_{i,j} \in \{0,1\} \qquad \text{for all } 1 \le i < j \le n\,. \tag{6.6}$$

In order to perform a branching in the TSP, the easiest way is to select some edge $e = (i,j)$ and set $P = P_e^{(+)} \dot\cup P_e^{(-)}$, where $P_e^{(+)}$ contains the additional condition that the tour contains the edge e (i.e., $x_{i,j} = 1$), and $P_e^{(-)}$ the condition that the tour does not contain e (i.e., $x_{i,j} = 0$). Iterating the branching, we see that every node in the search tree can be written as a problem of the form $P_{X,Y}$ with $X, Y \subset \{(i,j) : 1 \le i < j \le n\}$, $X \cap Y = \emptyset$, where X denotes the set of all those edges that we include in the admissible tours, and Y the edges we exclude.

Each problem $P_{X,Y}$ can again be written as TSP with a modified cost function $c^{(X,Y)}$. To that end define $C := 1 + \sum_{i<j} c_{i,j}$ and let

$$c_{i,j}^{(X,Y)} := \begin{cases} c_{i,j} & \text{if } (i,j) \in X\,, \\ c_{i,j} + C & \text{if } (i,j) \notin (X \cup Y)\,, \\ c_{i,j} + 2C & \text{if } (i,j) \in Y\,. \end{cases}$$

Then a tour for the problem $P_{X,Y}$ satisfies the constaints that it contains every edge in X and no edge in Y, if and only if its modified weight is strictly smaller than $(n+1-|X|)C$. In addition, in this case, the original and the modified cost of the tour differ by precisely $(n-|X|)C$.

Because of these considerations, if we want to apply a branch-and-bound method for the solution of the TSP, we only have to find an efficient method for obtaining good lower bounds for the value of a TSP. A simple LP-relaxation, that is, the solution of the optimisation problem respecting the bounds (6.5) but not the integrality condition (6.6), is not advisable because of the excessively large number of inequalities in (6.5).

Definition 6.1.1. Let G be a complete graph with vertex set $V = \{1,\dots,n\}$. A 1-*tree* in G is a graph consisting of a spanning tree of the vertices $\{2,\dots,n\}$ and two edges between the vertex 1 and the vertex set $\{2,\dots,n\}$. ■

It is easy to see that every Hamiltonian circuit is a 1-tree. Conversely, a 1-tree is a Hamiltonian circuit, if and only if the degree of every vertex is precisely 2. Moreover, it turns out that the family of 1-trees can be easily characterised:

Lemma 6.1.2. *The set of vectors $x = (x_{i,j})_{i,j}$ with $x_{i<j} \in \{0,1\}$ describes a 1-tree, if and only if*

$$\begin{aligned} &\sum_{\substack{i,j=1\\ i<j}}^{n} x_{i,j} = n\,, \qquad \sum_{k=2}^{n} x_{1,k} = 2\,, \\ &\sum_{i<j\in I} x_{i,j} \le |I| - 1 \qquad \textit{for all } \emptyset \ne I \subseteq \{2,\dots,n\}\,. \end{aligned} \tag{6.7}$$

A similar characterisation of Hamiltonian circuits is also available:

Lemma 6.1.3. *The set of vectors $x = (x_{i,j})_{i<j}$ with $x_{i,j} \in \{0,1\}$ describes a Hamiltonian circuit, if and only if*

$$\begin{aligned} &\sum_{\substack{i,j=1\\ i<j}}^{n} x_{i,j} = n\,, \qquad \sum_{k=1}^{i-1} x_{k,i} + \sum_{k=i+1}^{n} x_{i,k} = 2 \textit{ for all } i = 1,\dots,n\,, \\ &\sum_{i<j\in I} x_{i,j} \le |I| - 1 \qquad \textit{for all } \emptyset \ne I \subsetneq \{1,\dots,n\}\,. \end{aligned}$$

Thus, the theory of Lagrangean relaxation implies the following result (note that we relax a set of equations, not inequalities; thus the Lagrange parameters may also be negative):

Theorem 6.1.4 (Held and Karp). *Consider a TSP with weights $c_{i,j} > 0$. For every $\lambda = (\lambda_2, \ldots, \lambda_n) \in \mathbb{R}^{n-1}$ the value*

$$\begin{aligned} &L(c,\lambda) \\ &:= \min\Big\{\sum_{i<j} c_{i,j}x_{i,j} + \sum_{i=2}^{n} \lambda_i\Big(\sum_{k=1}^{i-1} x_{k,i} + \sum_{k=i+1}^{n} x_{i,k} - 2\Big) : (x_{i,j})_{i<j} \text{ satisfies } (6.7)\Big\}\end{aligned}$$

is a lower bound for the value of the problem.

Define moreover

$$HK(c) := \max\big\{L(c,\lambda) : \lambda \in \mathbb{R}^{n-1}\big\}\,.$$

Then

$$\begin{aligned} HK(c) = \min\Big\{ &c^T x : 0 \le x \le 1\,, \sum_{k=1}^{i-1} x_{k,i} + \sum_{k=i+1}^{n} x_{i,k} = 2 \text{ for all } 1 \le i \le n\,, \\ &\sum_{i<j\in I} x_{i,j} \le |I| - 1 \text{ for all } \emptyset \ne I \subseteq \{2,\ldots,n\}\Big\}\,. \end{aligned}$$

In particular, $HK(c)$ provides a lower bound for the value of the TSP.

Theorem 6.1.5 (Wolsey). *If the weigths $c_{i,j}$ satisfy the triangle inequality*

$$c_{i,j} + c_{j,k} \ge c_{i,k} \qquad \text{for all } i,j,k,$$

that is, the TSP is metric, *then $HK(c)$ is at least $2/3$ of the value of the TSP.*

In order to take advantage of Theorem 6.1.4, one has to be able to compute the value $L(c,\lambda)$ fast for different values of $\lambda \in \mathbb{R}^{n-1}$. To that end, define for arbitrary $\lambda_1 \in \mathbb{R}$ the modified cost

$$c_{i,j}^{(\lambda)} := c_{i,j} + \lambda_i + \lambda_j\,.$$

Then computing $L(c,\lambda)$ is equivalent to finding a minimal 1-tree with respect to the modified weight $(\tilde{c}_{i,j})_{i<j}$. Indeed, define

$$M(c,\lambda) := \min\Big\{\sum_{i<j} c_{i,j}^{(\lambda)} x_{i,j} : (x_{i,j})_{i,j} \text{ satisfies } (6.7)\Big\}\,. \tag{6.8}$$

Then

$$M(c,\lambda) = L(c,\lambda) + 2\sum_{i=1}^{n} \lambda_i\,.$$

Moreover, the minimisation problem in (6.8) can be solved by finding a minimal spanning tree on the vertices $\{2,\ldots,n\}$, and then connecting the vertex 1 with that tree by simply adding two edges of minimal weight from 1 to $\{2,\ldots,n\}$. For the computation of a minimal spanning tree, a greedy algorithm (cf. Section 5.1) can be applied, with which the problem can be solved in $O(n^2)$ time. For the maximisation of the Lagrange functional $L(c,\lambda)$, one can then use the sugradient method presented in Proposition 4.2.3.

6.2 Dynamical Programming

In many cases of discrete optimisation problems, it is possible to compute the optimal solution recursively by solving smaller sub-problems and then assemble these partial solutions to an optimum of the full problem.

6.2.1 Shortest Paths

A classical example is that of finding the shortest paths in a weighted graph from one given vertex s to all the other vertices. Then, if the shortest path between s and t passes through the vertex r, necessarily, the sub-graph starting in s and ending in r has to be the shortest path from s to r. One can take advantage of this observation, if one considers the sub-problems that consist of finding the shortest paths *consisting of at most k edges*, $1 \leq k \leq |V| - 1$, from the vertex s to all the other vertices. Then it is easy to compute the shortest paths of at most $k+1$ edges given those of at most k edges: One scans through all the edges (r, t) of the graph, tests whether attaching this edge to the shortest path from s to r of at most k edges would decrease the length of the currently shortest path from s to t, and, in case it would, replaces the shortest path accordingly. Since every shortest path may contain at most $|V| - 1$ edges (at least, if the weights of the edges are all positive), this algorithm will find all the shortest paths after at most $|V| - 1$ iterations.

```
Data: A (directed) graph $G = (V, E)$, positive weights $w(e)$, $e \in E$, and
      a vertex $s \in V$;
Result: For each vertex $t \in V$ reachable from $s$ the length $l(t)$ of the
        shortest path from $s$ to $t$ and the previous vertex $p(t)$ lying on
        this path;
Initialisation: Set $l(s) = 0$ and $l(t) = +\infty$ for all $t \in V \setminus \{s\}$;
for $i = 1, \dots, |V| - 1$ do
    for each $e = (r, t) \in E$ do
        if $l(t) > l(r) + w(e)$ then
            $l(t) \leftarrow l(r) + w(e)$;
            $p(t) \leftarrow r$;
        end
    end
end
```

Algorithm 11: Shortest paths in a directed graph

Remark 6.2.1. In fact, the positivity of the weights is not required for the algorithm to work. It yields the correct result also in case the graph G contains some negative edges, as long as there are no cycles in G of total negative length.■

6.2.2 The Knapsack Problem

Similar ideas can also be applied to the knapsack problem. Recall that here the task is to find a subset $S \subset \{1, \dots, n\}$ maximising $\sum_{j \in S} c_j$ subject to the constraint $\sum_{j \in S} a_j \leq b$. Assume now that all the costs c_j are positive integers, and define for $0 \le i \le n$ and $k \in \mathbb{N} \cup \{0\}$ the number $J(i,k)$ as

$$J(i,k) = \min\Big\{\sum_{j\in S} a_j : S \subset \{1,\dots,i\} \text{ and } \sum_{j\in S} c_j = k\Big\}\ .$$

That is, $J(i,k)$ is the minimal weight of a subset of $\{1,\dots,i\}$ such that the total cost of this set equals precisely k. Then

$$\min\Big\{\sum_{j\in S} c_j : S \subset \{1,\dots,n\},\ \sum_{j\in S} a_j \le b\Big\} = \max\Big\{k \in \mathbb{N} : J(n,k) \le b\Big\}\ .$$

Moreover, $J(i,\cdot)$ can be computed from $J(i-1,\cdot)$ by

$$J(i,k) = \min\big\{J(i-1,k),\ J(i-1,k-c_i) + a_i\big\}\ .$$

Thus one arrives at an algorithm of complexity $O(nC)$, where C is an a–priori estimate of the maximal value of the knapsack problem—for instance one can choose $C = \sum_j c_j$. A better bound can usually be obtained using the Best–in–greedy Algorithm 2 with weights $w_j = c_j/a_j$: One can show that the value C_{greedy} of the greedy solution is at least half the maximal value of the knapsack problem. Therefore setting $C := 2C_{\text{greedy}}$ provides a guaranteed upper bound.

Remark 6.2.2. Note that, in case one sets $C := \sum_{j=1}^n c_j$, in Algorithm 12 the maximal weight b is only needed for the assembly of the solution, but not for the definitions of J and s. Therefore the algorithm can be very efficient, if one wants to compute the optimal values for different maximal weights. ■

Remark 6.2.3. One may also exchange the roles of c and a in the algorithm and base the dynamical programme on the function

$$\tilde{J}(i,k) = \max\Big\{\sum_{j\in S} c_j : S \subset \{1,\dots,i\} \text{ and } \sum_{j\in S} a_j = k\Big\}\ .$$

Then one arrives at an algorithm of complexity $O(nb)$. ■

```
Data: $n \in \mathbb{N}$, values $c_i \in \mathbb{N}$, $1 \leq i \leq n$, weights $a_i \in \mathbb{N}$, $1 \leq i \leq n$, and a
      maximal weight $b \in \mathbb{N}$;
Result: A set $S \subset \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i \leq b$ and $\sum_{i \in S} c_i$ is
        maximal;
Initialisation: Choose an upper bound $C \in \mathbb{N}$ of the value of the
                maximisation problem;
Set $J(0,0) := 0$ and $J(0,k) := +\infty$ for $k = 1, \ldots, C$;
for $i = 1, \ldots, n$ do
    for $j = 0, \ldots, C$ do
        if $j < c_i$ then
            Set $s(i,j) := 0$ and $J(i,j) := J(i-1,j)$;
        else if $J(i-1, j-c_i) + a_i \leq J(i-1,j)$ then
            Set $s(i,j) := 1$ and $J(i,j) := J(i-1, j-c_i) + a_i$;
        else
            Set $s(i,j) := 0$ and $J(i,j) := J(i-1,j)$;
        end
    end
end
Let $j = \max\{0 \leq k \leq C : J(n,k) \leq b\}$;
Set $S := \emptyset$;
for $i = n, \ldots, 1$ do
    if $s(i,j) = 1$ then
        $S \leftarrow S \cup \{i\}$;
        $j \leftarrow j - c_i$;
    end
end
```

Algorithm 12: Dynamical programming for the knapsack problem with integer coefficients

Appendix A

Graphs

In the following, we recall the main notions from graph theory that are used in these notes.

A.1 Basics

A (undirected) *graph* is a triple $G = (V, E, \Psi)$, where V and E are finite sets (the *vertices* and the *edges* of the graph), and $\Psi \colon E \to \bigl\{W : W \subset V\bigr\}$ is a mapping satisfying $\#\Psi(e) = 2$ for every $e \in E$. That is, the mapping Ψ assigns to every edge E precisely two vertices. If two edges are assigned the same vertices, that is, if $\Psi(e) = \Psi(\tilde{e})$ for some $e \neq \tilde{e} \in E$, then these edges are called *parallel*. Typically, we assume that the graphs we consider do not contain parallel edges (they are *simple*). In this case, one can (and we do) identify the edge e with its image $\Psi(e) \subset \bigl\{W \subset V : \#W = 2\bigr\}$ and write $G = (V, E)$, omitting the mapping Ψ. Two vertices v and w are *adjacent*, if $\{v, w\} \in E$; the edge $\{v, w\}$ is then said to *join* v and w. If v is a vertex and $e = \{v, w\}$ an edge, then e is said to be *incident* with v. By $\delta(v)$ we denote the set of edges incident with v. The *degree* of v is the number of edges incident with v, that is, the degree of v equals $\#(\delta(v))$. More general, for every subset $X \subset V$ we define

$$\delta(X) := \bigl\{\{v, w\} \in E : v \in X,\ w \notin X\bigr\} .$$

A *directed graph* is a triple $G = (V, E, \Psi)$ with $\Psi \colon E \to \bigl\{(v, w) \in V \times V : v \neq w\bigr\}$. As in the case of undirected graphs, two edges $e \neq \tilde{e}$ are called parallel if $\Psi(e) = \Psi(\tilde{e})$. If a directed graph contains no parallel edges, then we again identify edges with their images and write $G = (V, E)$ with $E \subset V \times V$. Note that the edges (v, w) and (w, v) with $v \neq w \in V$ are *not* parallel. Similarly as for undirected graphs, we say that two vertices $v \neq w$ are adjacent, if either (v, w) or (w, v) is an edge (note that in directed graphs these two edges are different). We say that the edge (v, w) *leaves* v and *enters* w. We define

$$\begin{aligned}
\delta^+(v) &:= \bigl\{e \in E : e = (v, w) \text{ for some } w \in V\bigr\} ,\\
\delta^-(v) &:= \bigl\{e \in E : e = (w, v) \text{ for some } w \in V\bigr\} ,
\end{aligned}$$

the sets of edges leaving and entering v, repectively. The *out-degree* of v is defined as $\#(\delta^+(v))$; the *in-degree* as $\#(\delta^-(v))$. More general, for every subset

$X \subset V$ we define

$$\begin{aligned}\delta(X)^+ &:= \big\{(v,w) \in E : v \in X,\ w \notin X\big\}\,,\\ \delta(X)^- &:= \big\{(w,v) \in E : v \in X,\ w \notin X\big\}\,.\end{aligned}$$

If $G = (V, E)$ is a directed graph, then the *underlying undirected graph* is the graph $G' = (V, E')$ with the same set of vertices and edges of the form $\{v, w\}$ with $(v, w) \in E$.

Let $G = (V, E)$ be a (directed) graph. A *sub-graph* of G is a (directed) graph $G' = (V', E')$ with $V' \subset V$ and $E' \subset E$. We say that G' is the sub-graph of G *induced by* V', if

$$E' = \big\{\{u,v\} : u,\, v \in V' \text{ and } \{u,v\} \in E\big\}$$

or

$$E' = \big\{(u,v) : u,\, v \in V' \text{ and } (u,v) \in E\big\}\,.$$

That is, the sub-graph G' of G is induced by V' if it contains all the edges of G that join vertices in V'. We say that the sub-graph is *spanning*, if $V' = V$.

We say that a graph $G = (V, E)$ is *complete*, if either $E = \big\{\{v,w\} : v \neq w \in V\big\}$ in the undirected case, or $E = V \times V \setminus \big\{(v,v) : v \in V\big\}$ in the directed case. That is, a complete graph is a maximal simple graph for a given vertex set; the addition of any edge would destroy its simplicity.

A.2 Paths, Circuits, and Trees

In the following we always assume that $G = (V, E)$ is either a directed or an undirected graph.

A *walk* in G is a sequence

$$W = (v_1, e_1, v_2, e_2, v_3, \ldots, v_k, e_k, v_{k+1})$$

such that $v_i \in V$, $e_i \in E$, and for every i we have $e_i = \{v_i, v_{i+1}\}$ if G is undirected or $e_i = (v_i, v_{i+1})$ (that is, the edges lead from one vertex to the next vertex). The walk is closed, if $v_1 = v_{k+1}$.

A *path* in G is a walk where all the visited vertices are different. We often identify a path $(v_1, e_1, v_2, \ldots, v_k, e_k, v_{k+1})$ with the subgraph

$$P := (\{v_1, \ldots, v_{k+1}\}, \{e_1, \ldots, e_k\})$$

of G.

A *circuit* or *cycle* is a closed walk where all the visited vertices apart from the last and first one are different. As in the case of paths, we identify a circuit $(v_1, e_1, v_2, \ldots, v_k, e_k, v_1)$ with the subgraph

$$C := (\{v_1, \ldots, v_k\}, \{e_1, \ldots, e_k\})$$

of G, thus forgetting about the starting vertex.

A *Hamiltonian path* is a path that visits all the vertices in G (it is spanning); similarly, a *Hamiltonian circuit* is a circuit that visits all the vertices in G.

The *length* of a path or a circuit is the number of its edges. The *distance* between two vertices v and w, denoted by $\operatorname{dist}_G(v,w)$ or simply $\operatorname{dist}(v,w)$, is the length of the shortest path from v to w (or, equivalently, the length of the shortest walk from v to w). If no path from v to w exists, then we set $\operatorname{dist}(v,w) := +\infty$. If G is an undirected graph, then $\operatorname{dist}(v,w) = \operatorname{dist}(w,v)$; for a directed graph, this symmetry need not hold. Note that we always have that $\operatorname{dist}(v,v) = 0$ for all $v \in V$, because $P = (\{v\},\emptyset)$ is also a path in G. For fixed $v \in V$, the vertices w for which $\operatorname{dist}(v,w) < +\infty$ are called *reachable* from v.

In many optimisation problems, we are in addition given a cost function $c\colon E \to \mathbb{R} \cup \{+\infty\}$. In this case, we define

$$\operatorname{dist}_c(v,w) := \min\Bigl\{\sum_{e\in F} c(e) : P = (W,F) \text{ is a path from } v \text{ to } w\Bigr\}.$$

In addition, we extend the function c to the class of all subsets of E setting

$$c(F) := \sum_{e\in F} c(e).$$

With this notation we have

$$\operatorname{dist}_c(v,w) := \min\bigl\{c(P) : P \text{ is a path from } v \text{ to } w\bigr\}.$$

An undirected graph G is called *connected* if all vertices $w \in V$ are reachable from some (or, equivalently, every) vertex $v \in V$; else it is called *disconnected.* The maximal connected subgraphs of G are called the *connected components* of G.

An undirected graph is a *forest*, if it contains no circuits as subgraphs; a *tree* is a connected forest. A *leaf* in a tree is a vertex of degree 1.

Proposition A.2.1. *Let G be an undirected graph with n vertices. The following statements are equivalent:*

1. *G is a tree.*
2. *G has $n-1$ edges and contains no circuits.*
3. *G has $n-1$ edges and is connected.*
4. *If $v \neq w$ are vertices in G, then there exists a unique path from v to w in G.*
5. *G is connected, but the removal of any edge makes G disconnected.*
6. *G contains no circuit, but the addition of any edge not already contained in G creates a circuit.*

Index