Chapter 2

Algorithms and Complexity

In this chapter we discuss some basic concepts and solution methods for combinatorial optimization problems. In a **combinatorial (discrete) optimization problem** we are given a finite set \( S \) and an objective function \( c : S \rightarrow \mathbb{R} \). While for minimization problems we have to find a solution \( s^* \in S \) with \( c(s^*) \leq c(s) \) for all \( s \in S \), for maximization problems we have to find a solution \( s^* \in S \) with \( c(s^*) \geq c(s) \) for all \( s \).

In Section 2.1 a short introduction to complexity theory is given, which is useful to decide whether an optimization problem is easy or hard to solve. Examples for easy problems are shortest path problems, which are discussed in Section 2.2, and optimization problems which can be formulated as linear programs. Section 2.3 contains an introduction to linear programming, Section 2.4 introduces Lagrangian relaxation. Section 2.5 is devoted to network flow problems.

For hard optimization problems exact algorithms (which always determine an optimal solution) and approximation algorithms (which only provide approximate solutions) are distinguished. Exact algorithms like branch-and-bound, constraint programming, and dynamic programming are described in Sections 2.6, 2.7, and 2.9, respectively. In Section 2.8 the satisfiability problem is introduced; in Section 2.10 local search procedures, genetic algorithms and ant colony optimization systems are presented as examples for approximation algorithms.

2.1 Easy and Hard Problems

When we consider scheduling problems (or more generally combinatorial optimization problems) an important issue is the complexity of a problem. For a new problem we often first try to develop an algorithm which solves the problem in an efficient way. If we cannot find such an algorithm, it may be helpful to prove that the problem is NP-hard, which implies that with high probability no efficient algorithm exists. In this section we review the most important aspects of complexity theory which allow us to decide whether a problem is “easy” or “hard”.

2.1.1 Polynomially solvable problems

The efficiency of an algorithm may be measured by its running time, i.e. the number of steps it needs for a certain input. Since it is reasonable that this number is larger for a larger input, an efficiency function should depend on the input size. The size of an input for a computer program is usually defined by the length of a binary encoding of the input. For example, in a binary encoding an integer number $a$ is represented as binary number using $\log_2 a$ bits, an array with $m$ numbers needs $m \log_2 a$ bits when $a$ is the largest number in the array. On the other hand, in a so-called unary encoding a number $a$ is represented by $a$ bits.

Since in most cases it is very difficult to determine the average running time of an algorithm for an input with size $n$, often the worst-case running time of an algorithm is studied. For this purpose, the function $T(n)$ may be defined as an upper bound on the running time of the algorithm on any input with size $n$. Since often it is also difficult to give a precise description of $T(n)$, we only consider the growth rate of $T(n)$ which is determined by its asymptotic order. We say that a function $T(n)$ is $O(g(n))$ if constants $c > 0, n_0 \in \mathbb{N}$ exist such that $T(n) \leq cg(n)$ for all $n \geq n_0$. For example, the function $T_1(n) = 37n^3 + 4n^2 + n$ is $O(n^3)$, the function $T_2(n) = 2^n + n^{100} + 4$ is $O(2^n)$.

If the running time of an algorithm is bounded by a polynomial function in $n$, i.e. if it is $O(n^k)$ for some constant $k \in \mathbb{N}$, the algorithm is called a polynomial-time algorithm. If the running time is bounded by a polynomial function in the input size of a unary encoding, an algorithm is called pseudo-polynomial. For example, an algorithm with running time $O(n^2a)$ is pseudo-polynomial if the input size of a binary encoding is $O(n \log a)$.

Since exponential functions grow much faster than polynomial functions, exponential-time algorithms cannot be used for larger problems. If we compare two algorithms with running times $O(n^3)$ and $O(2^n)$ under the assumption that a computer can perform $10^9$ steps per second, we get the following result: While for $n = 1000$ the first algorithm is still finished after one second, the second algorithm needs already 18 minutes for $n = 40$, 36 years for $n = 60$ and even 374 centuries for $n = 70$.

For this reason a problem is classified as “easy” (tractable) if it can be solved by a polynomial-time algorithm. In order to classify a problem as “hard” (intractable), the notion of NP-hardness was developed which will be discussed in the next subsection.

2.1.2 NP-hard problems

In order to define NP-hardness at first some preliminary definitions have to be given. A problem is called a decision problem if its answer is only “yes” or “no”. A famous example for a decision problem is the so-called partition