Chapter 1

Introduction

1.1 Examples

We start with some examples of combinatorial optimization problems.

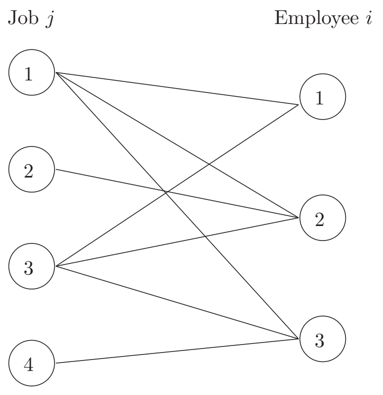
Example 1.1. The following problem is called the Knapsack problem. We are given an amount of C Euro and wish to invest it among a set of n options. Each such option i has cost c_i and profit p_i. The goal is to maximize the total profit.

Consider $C = 100$ and the following cost-profit table:

Option	Cost	Profit
1	100	150
2	1	2
3	50	55
4	50	100

Our choice of purchased options must not exceed our capital C. Thus the feasible solutions are $\{1\}, \{2\}, \{3\}, \{4\}, \{2,3\}, \{2,4\}, \{3,4\}$. Which is the best solution? We evaluate all possibilities and find that $\{3,4\}$ give 155 altogether which maximizes our profit.

Example 1.2. Another example is a Load Balancing problem: We have m employees, where each one has certain qualifications. Furthermore, we have a set of n jobs that need to be done and each job j has a processing time p_j. However, maybe not every employee is qualified to work on a certain job j. For each job j we introduce a set S_j of the employees that are eligible to work on that particular job. The following diagramm visualizes the sets S_j: for each job j (on the left hand side) we see which employee (on the right hand side) is able to work on that job.



We can formulate our problem with the following mathematical program. We use the variables $x_{i,j} \in \{0,1\}$ that indicate if employee i is assigned to job j. We want to minimize the time until all jobs are finished.

$$\begin{array}{llll}
\text{minimize} & \displaystyle\max_{i=1,\ldots,m} \sum_{j=1}^{n} p_j x_{i,j} & & \text{“minimize finishing time”} \\
\text{subject to} & \displaystyle\sum_{i \in S_j} x_{i,j} = 1 & j = 1,\ldots,n & \text{“each job gets done”} \\
& x_{i,j} \in \{0,1\} & i = 1,\ldots,m,\ j = 1,\ldots,n & \text{“assignment”}
\end{array}$$

Example 1.3. Many combinatorial optimization problems, like the ones above can be formulated in terms of a Integer Linear Program (ILP). Let $A = (a_{i,j})_{i=1,\ldots,m,j=1,\ldots,n} \in \mathbb{R}^{m,n}$ be a matrix and let $b = (b_i)_{i=1,\ldots,m} \in \mathbb{R}^m$ and $c = (c_j)_{j=1,\ldots,n} \in \mathbb{R}^n$ be vectors. Further let $x = (x_j)_{j=1,\ldots,n} \in \mathbb{Z}^n$ be variables that are allowed to take *integral* values, only. Our objective function is to minimize $c^\top x$ subject to $Ax \leq b$. More explicitly

$$\begin{array}{llll}
\text{minimize} & \displaystyle\sum_{j=1}^{n} c_j x_j & & \text{“objective function”} \\
\text{subject to} & \displaystyle\sum_{j=1}^{n} a_{i,j} x_j \leq b_i & i = 1,\ldots,m & \text{“constraints”} \\
& x_j \in \mathbb{Z} & j = 1,\ldots,n,\ j = 1,\ldots,n & \text{“integrality”}
\end{array}$$

Solving an ILP is in general NP-hard. However, we will often replace the constraints $x_j \in \mathbb{Z}$ with $x_j \in R$. This is then called a *relaxation* as a Linear Program (LP) and can be solved in polynomial time. Of course, such a solution is in general not feasible for the ILP, but we can sometimes “turn” it into a feasible solution, which is not “too bad”.

1.2 Combinatorial Optimization Problems

An instance of a *combinatorial optimization problem* (COP) can formally be defined as a tuple $I = (U, P, \mathrm{val}, \mathrm{extr})$ with the following meaning:

U	the *solution space* (on which val and S are defined),
P	the *feasibility predicate*,
val	the *objective function* $\mathrm{val} : U \to \mathbb{R}$,
extr	the *extremum* (usually max or min).

The feasibility predicate P induces a set:

S the set of *feasible solutions*: $S = \{X \in U : X \text{ satisfies } P\}$.

Our goal is to find a feasible solution where the desired extremum of val is attained. Any such solution is called an *optimum solution*, or simply an *optimum*. U and S are usually not given explicitly, but implicitly.

Let us investigate the problem in Example 1.1 in with this formalism.

$$\begin{aligned}
U &= 2^{\{1,2,3,4\}},\\
P &= \text{“total cost is at most } C\text{”, i.e., } X \in S \text{ if } \sum_{i \in X} c_i \le C\\
S &= \{\{1\},\{2\},\{3\},\{4\},\{2,3\},\{2,4\},\{3,4\}\},\\
\mathrm{val} &= \begin{cases} U \to \mathbb{R} \\ X \mapsto \sum_{i\in X} p_i, \end{cases}\\
\mathrm{extr} &= \max .
\end{aligned}$$

The optimum solution here is $\{3,4\}$ with value 155.

A central problem around combinatorial optimization is that it is often in principle possible to find an optimum solution by enumerating the set of feasible solutions, but this set mostly contains “too many” elements. This phenomenon is called *combinatorial explosion*.

1.3 Algorithms and Approximation

Many problems in combinatorial optimization can be solved by using an appropriate algorithm. Informally, an *algorithm* is given a (valid) input, i.e., a description of an instance of a problem and computes a solution after a finite number of “elementary steps”. The number of bits used to describe an input I is called the (binary) *length* or *size* of the input and denoted $\mathrm{size}(I)$.

Let $t : \mathbb{N} \to \mathbb{R}$ be a function. We say that an algorithm *runs* in time $O(t)$ if there is a constant α such that the algorithm uses at most $\alpha t(\mathrm{size}(I))$ many elementary steps to compute a solution given any input I. An algorithm is called *polynomoial time* if $t : n \mapsto n^c$ for some constant c. This contrasts *exponential time* algorithms where $t : n \mapsto c^n$ for some constant $c > 1$.

Because the running times of exponential time algorithms grow rather rapidly as the input size grows, we are mostly interested in polynomial time algorithms. Of course, we desire to find an optimum solution for any given COP in polynomial time. Unfortunately this is not always possible as many COPs are NP-hard. (It is widely believed that no polynomial time algorithm exists that solves some NP-hard COP optimally on every instance.) Thus our goal is a find “good” solutions in polynomial time.

Let $\Pi = \{I_1, I_2, \dots\}$ be a set of instances of a COP, where each $I \in \Pi$ is of the form $I = (U, P, S, \mathrm{val}, \mathrm{extr})$. For any $I \in \Pi$, let $\textsc{opt}(I) = \mathrm{extr}_{X\in S(I)}\mathrm{val}(X)$ denote the respective optimum value. An *approximation algorithm* $\textsc{alg}$ for Π is a polynomial time algorithm that computes some solution $X \in S(I)$ for every instance $I \in \Pi$. The respective value obtained is denoted $\textsc{alg}(I) = \mathrm{val}(X)$. The *approximation ratio* of $\textsc{alg}$ on an instance I is defined by

$$\rho_{\textsc{alg}}(I) = \frac{\textsc{alg}(I)}{\textsc{opt}(I)}.$$

The algorithm $\textsc{alg}$ is a *ρ-approximation* algorithm if

$$\begin{aligned}
\rho_{\textsc{alg}}(I) &\le \rho \qquad \text{for all } I \in \Pi \text{ and extr} = \min,\\
\rho_{\textsc{alg}}(I) &\ge \rho \qquad \text{for all } I \in \Pi \text{ and extr} = \max.
\end{aligned}$$