A. Introduction to $\mathcal{NP}$-Completeness of Knapsack Problems

The reader may have noticed that for all the considered variants of the knapsack problem, no polynomial time algorithm have been presented which solves the problem to optimality. Indeed all the algorithms described are based on some kind of search and prune methods, which in the worst case may take exponential time. It would be a satisfying result if we somehow could prove it is not possible to find an algorithm which runs in polynomial time, somehow having evidence that the presented methods are "as good as we can do". However, no proof has been found showing that the considered variants of the knapsack problem cannot be solved to optimality in polynomial time.

The theory of $\mathcal{NP}$-completeness gives us a framework for showing that it is very doubtful that a polynomial algorithm exists. Indeed, if we could find a polynomial algorithm for solving e.g. the subset sum problem, then we would also be able to solve numerous famous optimization problems like the *traveling salesman problem*, *general integer programming*, and we would even be able to efficiently find *mathematical proofs* of theorems, as stated in Cook [91]. A very comprehensive guide to the theory of $\mathcal{NP}$-completeness is found in the seminal book by Garey and Johnson [164]. Simplified introductions can be found in nearly any text book on combinatorial optimization or algorithms, e.g. in Cormen et al. [92], or Papadimitriou and Steiglitz [367]. A compendium on $\mathcal{NP}$-optimization problems can be found in Crescenzi and Kann [94].

A.1 Definitions

The following discussion is based on the assumption that problems which are solvable in polynomial time, somehow are *tractable*. As introduced in Section 1.5, we will use the big-Oh notation to describe the asymptotic running time of an algorithm. Although a running time of $O(n^{100})$ may not seem very attractive, polynomial running times have some nice properties which support this assumption. Moore's law [348] argues that the speed of computers is doubled every 18 months. For a polynomially solvable problem, each time the speed of the computer gets doubled, we will be able to solve problems which are larger by a multiplicative factor, while we only

H. Kellerer et al., *Knapsack Problems*

get an additive increase for problems demanding an exponential number of iterations. If for instance a problem is solvable in $O(n^3)$ then each time the speed of the computer is doubled, we may solve problems which are $\sqrt[3]{2}$ larger using the same computational time. If a problem is solvable in time $O(2^n)$ then each time the speed of the computer is doubled we may solve problems which are only a single decision variable larger within the same computational time.

We will restrict our discussion to *decision problems*, i.e. problems which may be answered by a "yes" or a "no". An optimization problem is easily transformed to a decision problem by comparing the solution value with a threshold value. For instance the knapsack problem in decision form asks whether a solution to the knapsack problem with objective value larger than t exits. Hence, the optimization and decision problems may be stated as:

$$\text{KP-OPTIMIZATION}(p,w,c) = \left\{ \begin{array}{l} \max \sum_{j=1}^{n} p_j x_j \\ \text{s.t.} \sum_{j=1}^{n} w_j x_j \leq c, \\ \quad x_j \in \{0,1\}, j = 1,\ldots,n \end{array} \right\} \tag{A.1}$$

$$\text{KP-DECISION}(p,w,c,t) = \left\{ \begin{array}{l} \text{there exists an } x \text{ with} \\ \sum_{j=1}^{n} p_j x_j \geq t, \\ \sum_{j=1}^{n} w_j x_j \leq c, \\ x_j \in \{0,1\}, j = 1,\ldots,n \end{array} \right\} \tag{A.2}$$

If we are able to solve the decision problem efficiently we are generally also able to solve the optimization problem efficiently. As an example we may solve KP-OPTIMIZATION by using *binary search* for the optimal solution value by a number of calls to KP-DECISION. Knowing that the optimal solution must be between $a := 0$ and $b := \sum_{j=1}^{n} p_j$, we make a call to KP-DECISION using the threshold value $t := \frac{a+b}{2}$. If we get the answer "yes" we set $a := t$ otherwise we set $b := t - 1$ and repeat the process. When $a = b$ we have the optimal solution value. The number of calls to KP-DECISION is bounded by $\log(\sum_{j=1}^{n} p_j)$ which is polynomial in the length of the input. If KP-DECISION could be solved in polynomial time we could also solve KP-OPTIMIZATION in polynomial time.

While KP-DECISION is a general problem, an *instance* of KP-DECISION is a concrete dataset $I = \{p_1,\ldots,p_n,w_1,\ldots,w_n,c\}$. This could e.g. be the following dataset

j	1	2	3	4
p_j	8	5	2	3
w_j	5	2	4	8

$c = 10,\ t = 9$ (A.3)

It is easy to see that the instance is a "yes"-instance since choosing item 1 and 3 gives a feasible solution with profit sum of 10, exceeding the threshold value t.

Since we measure the time and space complexity as a function of the *input size* $L(I)$, one must define some standards how the input size of an instance I is measured. We will use the natural assumption that all input data is written in binary form, and