Chapter 4. The knapsack problem

Section 4.1. Formulation

We are given a set of n items; each item i $(i = 1, \ldots, n)$ has a value $c_i > 0$ and a weight $a_i > 0$. A knapsack with (weight) capacity b has to be filled with items so as to maximize the total value of the items included in the knapsack. Without loss of generality, we assume that all weights a_i and values c_i are integral; due to the integrality of the weights, we can also assume that b is integral.

We formulate the knapsack problem (KS) by using the binary variables x_i $(i = 1, \ldots, n)$, where the outcome $x_i = 1$ signals that item i must be included in the knapsack and $x_i = 0$ signals that item i has to be left at home.

$$\text{(KS)} \qquad \max \sum_{i=1}^{n} c_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} a_i x_i \leq b,$$

$$\forall_{i \in \{1,\ldots,n\}} \; x_i \in \{0, 1\}.$$

We suppose that $\sum_{i=1}^{n} a_i > b$ and $a_i \leq b \;\; (i = 1, \ldots, n)$ to avoid trivialities.

Although the knapsack problem contains only one non-trivial linear constraint, there is no polynomial algorithm known that solves the problem to optimality. On the other hand, there is no proof that such an algorithm does not exist. There are two alternatives to cope with problems for which no efficient solution methods exist. The first one is to apply an algorithm with *exponential* running time to find an *optimal* solution. To solve the knapsack problem, we can apply the general techniques of *branch and bound* and *dynamic programming*. Both techniques enumerate all possible feasible solutions in a smart way. The second possibility is to apply a *polynomial* algorithm that constructs a good but *possibly non-optimal* solution. Such inexact algorithms are called *approximation algorithms*.

Section 4.2. The continuous knapsack problem

If we no longer demand a solution of the knapsack problem to be integral but allow for fractional values, then we get the *continuous knapsack problem* (CKS); this is a relaxation of the knapsack problem in the sense that the set of feasible solutions for the continuous knapsack problem contains the solution set of knapsack as a subset.

$$\text{(CKS)} \qquad \max \sum_{i=1}^{n} c_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} a_i x_i \leq b,$$

$$\forall_{i \in \{1,\ldots,n\}} \; 0 \leq x_i \leq 1 \; .$$

Consider any instance I_{KS} of the knapsack problem and the corresponding instance of the continuous knapsack problem I_{CKS}; I_{CKS} is called the linear programming relaxation, since the integrality constraints of KS are relaxed to linear constraints. Clearly, for the optimal values of I_{KS} and I_{CKS}, which are denoted by $z(I_{KS})$ and $z(I_{CKS})$, we have that $z(I_{KS}) \leq z(I_{CKS})$. Hence, $z(I_{CKS})$ forms an upper bound on $z(I_{KS})$.

Although the set of feasible solutions of I_{CKS} extends the set of feasible solution of I_{KS}, there is a straightforward greedy algorithm that solves CKS. Renumber the items according to non-increasing $\frac{c_i}{a_i}$ $(i = 1, \ldots, n)$, that is, $\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \ldots \geq \frac{c_n}{a_n}$. For this sequence we have that the most interesting items, that is, those with the highest value per unit of weight are numbered lowest. The greedy algorithm raises the values of the variables in the order $x_1, x_2, \ldots, x_n$.

Greedy algorithm CKS.

Input:	n items with weight $a_i > 0$, value $c_i > 0$, and a capacity $b > 0$.
Output:	An optimal solution $(x_1, \ldots, x_n)$ for the continuous knapsack.

STEP 1. All variables x_i $(i = 1, \ldots, n)$ are set equal to 0; the residual capacity $\bar{b}$ is set equal to b; $j \leftarrow 1$.

STEP 2. If $a_j \leq \bar{b}$, then $x_j \leftarrow 1$; otherwise, $x_j \leftarrow \bar{b}/a_j$. Set $\bar{b} \leftarrow \bar{b} - a_j x_j$; $j \leftarrow j + 1$.

STEP 3. If $\bar{b} > 0$, then go to STEP 2.

Theorem 4.1.
The greedy algorithm gives an optimal solution for CKS.

Proof.
Note that the algorithm fills the knapsack completely, since we assumed that $\sum_{i=1}^{n} a_i > b$ and $c_i > 0$ for all i; this implies that there exists a j, such that $1 = x_1 = \ldots = x_{j-1} > x_j \geq x_{j+1} = \ldots = 0$, where $x_{n+1} = 0$. We show the optimality of this solution by comparing it to any other feasible solution $y_1, \ldots, y_n$ of the continuous knapsack problem. Since all values c_i are positive, this solution can only be optimal if $\sum_{i=1}^{n} a_i y_i = b$. Let k be the smallest index such that $y_k < 1$, and let l be the smallest index with $k < l$ such that $y_l > 0$; note that such an l exists, unless the solution $y_1, \ldots, y_n$ is equal to the solution $x_1, \ldots, x_n$ obtained by the greedy algorithm. We will now increase y_k and decrease y_l, while keeping all other values equal, to obtain a new solution. Let $\varepsilon = \min\{a_k(1 - y_k), a_l y_l\} > 0$. Increase y_k by $\frac{\varepsilon}{a_k}$ and decrease y_l by $\frac{\varepsilon}{a_l}$. It is easily checked that this move yields a feasible solution with value no smaller than the value of the solution $y_1, \ldots, y_n$. Moreover, either y_k has become equal to 1, or y_l has become equal to 0. Repetition of this argument eventually yields the solution $x_1, \ldots, x_n$ obtained by the greedy algorithm. $\square$

Section 4.3. A branch and bound algorithm

The main characteristic of a branch and bound algorithm is that it partitions the solution set into subsets $S_1, \ldots, S_K$, which are described by a partial feasible solution, that is, some of the values of the variables $x_1, \ldots, x_n$ have been fixed. For example, we obtain a partition of the solution set into subsets S_1 and S_2 by *branching* with respect to the value of x_1: we request each solution in S_1 to have $x_1 = 1$ and each solution in S_2 to have $x_1 = 0$. We have that each subset itself boils down to a smaller knapsack problem. An upper bound UB_k on the outcome of the knapsack problem corresponding to S_k can therefore be calculated by means of the greedy algorithm for the continuous knapsack problem. A lower bound LB_k follows from rounding down the fractional variable, if it exists. If no fractional variable exists, then $LB_k = UB_k$.

Let z_F denote the value of the best feasible solution found so far; we have that $z_F \geq \max\{LB_k | k = 1, \ldots, K\}$. We call a pair (S_k, UB_k) *active* if $UB_k > z_F$: the subset S_k may contain a feasible solution with value greater than the value z_F of the current best solution. Hence, it might be wise to pay some extra attention to this subset. The pair (S_k, UB_k) is called *inactive* if $UB_k \leq z_F$: this subset of solutions is of no interest anymore, since it cannot contain a solution with value greater than z_F. Hence, we fathom this subset.

We implement our branch and bound algorithm as follows. We construct a list L that contains all active pairs (S_k, UB_k). The branching part of the algorithm consists of splitting a set S_k into two sets S_k^0 and S_k^1 according to our fixing the value of some variable x_j the value of which has not been fixed before; all solutions in the sets S_k^0 and S_k^1 are requested to have $x_j = 0$ and $x_j = 1$, respectively. Initially, L consists of the original problem, that is, there are no variables whose with fixed value. The solution of the continuous knapsack problem is used as an initial upper bound. We determine z_F by rounding down the value of the fractional variable.

As long as the list L of active pairs is not empty, we perform the following steps.

STEP 1. Choose from among the list of active pairs the one (S_k, UB_k) with maximal upper bound.

STEP 2. Split S_k in S_k^0 and S_k^1 according to the fractional variable x_i in the continuous solution.

STEP 3. Calculate UB_k^0 and UB_k^1 through the greedy algorithm CKS.

STEP 4. Round down the continuous solutions obtained in STEP 3 to obtain feasible solutions for KS; this gives the lower bounds LB_k^0 and LB_k^1.

STEP 5. If $LB_k^0 > z_F$, then set $z_F \leftarrow LB_k^0$; if $LB_k^1 > z_F$, then set $z_F \leftarrow LB_k^1$.

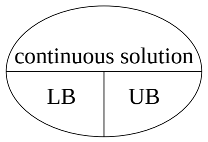
STEP 6. If $UB_k^0 > z_F$, then the pair (S_k^0, UB_k^0) is added to L; if $UB_k^1 > z_F$, then the pair (S_k^1, UB_k^1) is added to L.

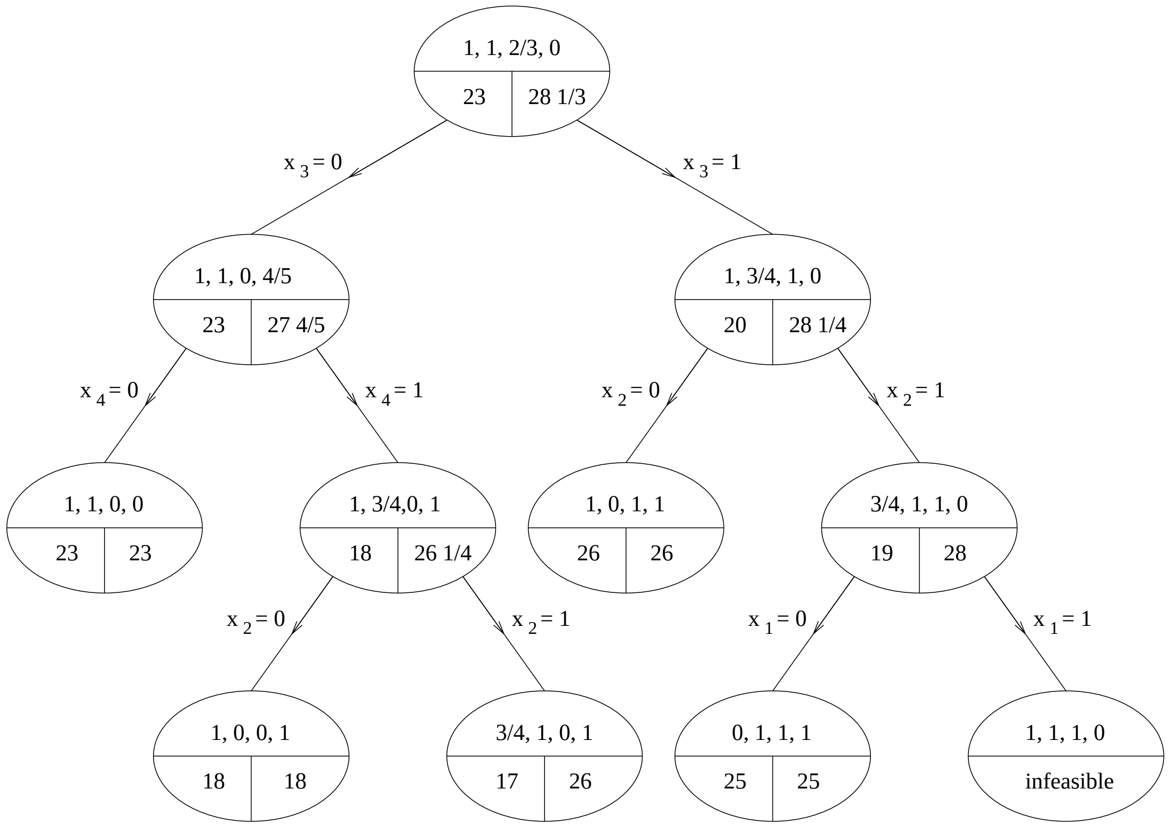
The branch and bound process is illustrated by displaying it in the form of a tree, where the active and inactive pairs (S_k, UB_k) form the leaves. Therefore the active sets are usually called nodes.

Example: 4 items; $b = 20$

item i	1	2	3	4
c_i	12	11	8	6
a_i	8	8	6	5

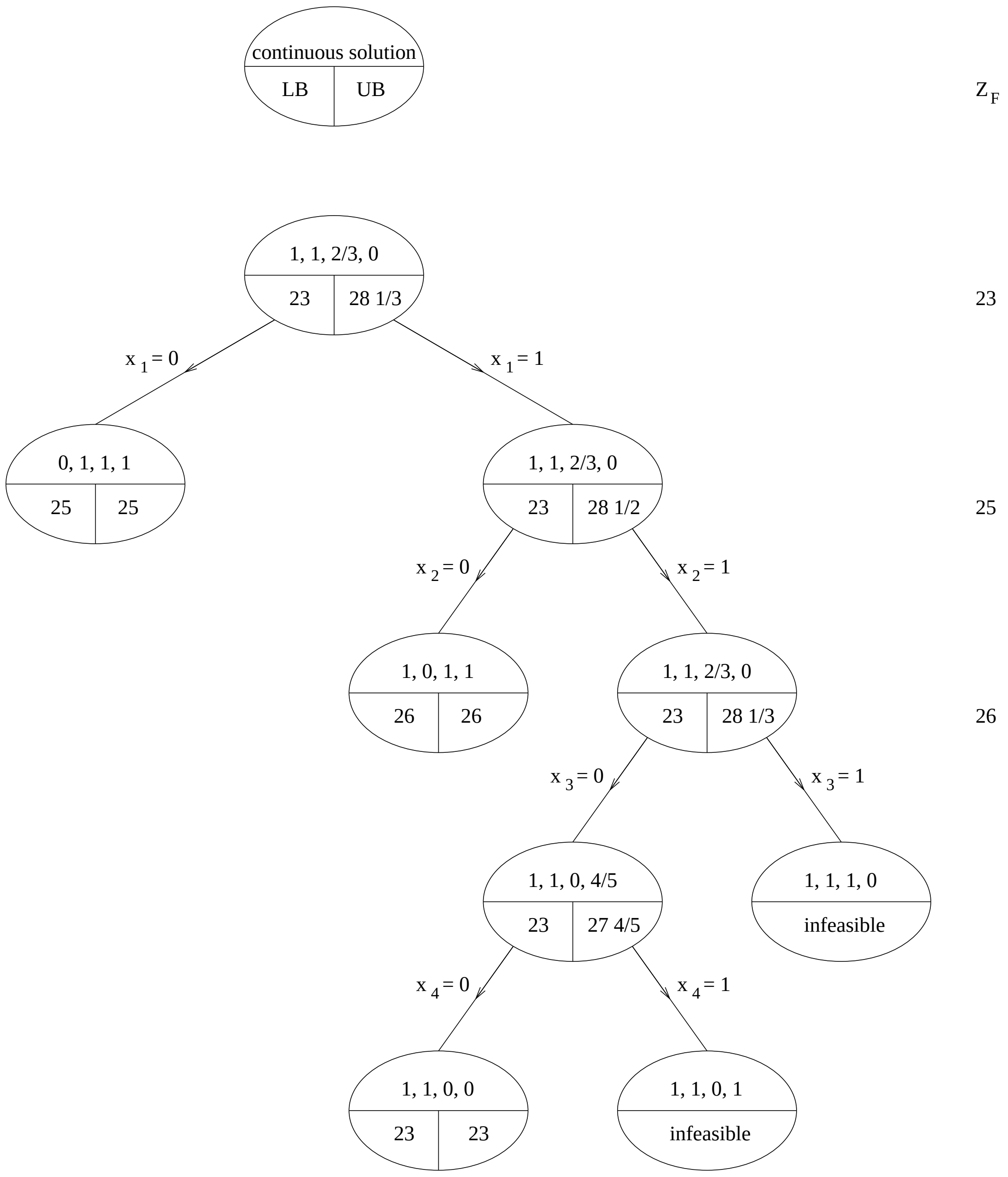
Each node in the tree contains the following information.





Note that, since the outcome of an optimal feasible solution for the knapsack problem is integral, we have that the outcome of the knapsack problem amounts to no more than the integral part of the outcome of the continuous knapsack problem. Hence, we can achieve a small improvement by declaring a pair (S_k, UB_k) inactive if $\lfloor UB \rfloor < z_F$. We can even go further by already fathoming a pair (S_k, UB_k) if $\lfloor UB \rfloor = z_F$: we are interested in finding one optimal solution only.

A more significant improvement is obtained by changing the branching rule, that is, the choice of the variable according to which the active subset is split. In the elaboration below, the variable with highest value per unit of weight is chosen, that is, the variable with the lowest index.



Section 4.4. Dynamic Programming

Another method that can be applied to solve the knapsack problem to optimality is *dynamic programming.* The idea behind the dynamic programming algorithm is that, if we know the maximal value $f_k(d)$ that can be gained by filling a knapsack of capacity d with the items $1,\ldots,k$ for all appropriate values of d, then we can readily determine the maximal values $f_{k+1}(d)$ for all appropriate values of d. As to which values of d are appropriate, we have that, since the weights a_i are all integral, only integral values of d are of interest; obviously, there is no use in considering values of d that are greater than b.

The values $f_k(d)$ can be calculated recursively from the values $f_{k-1}(d')$, with $d' = 0,\ldots,d$, as follows:

$$f_k(d) = \max\{f_{k-1}(d), f_{k-1}(d-a_k)+c_k\} \ .$$

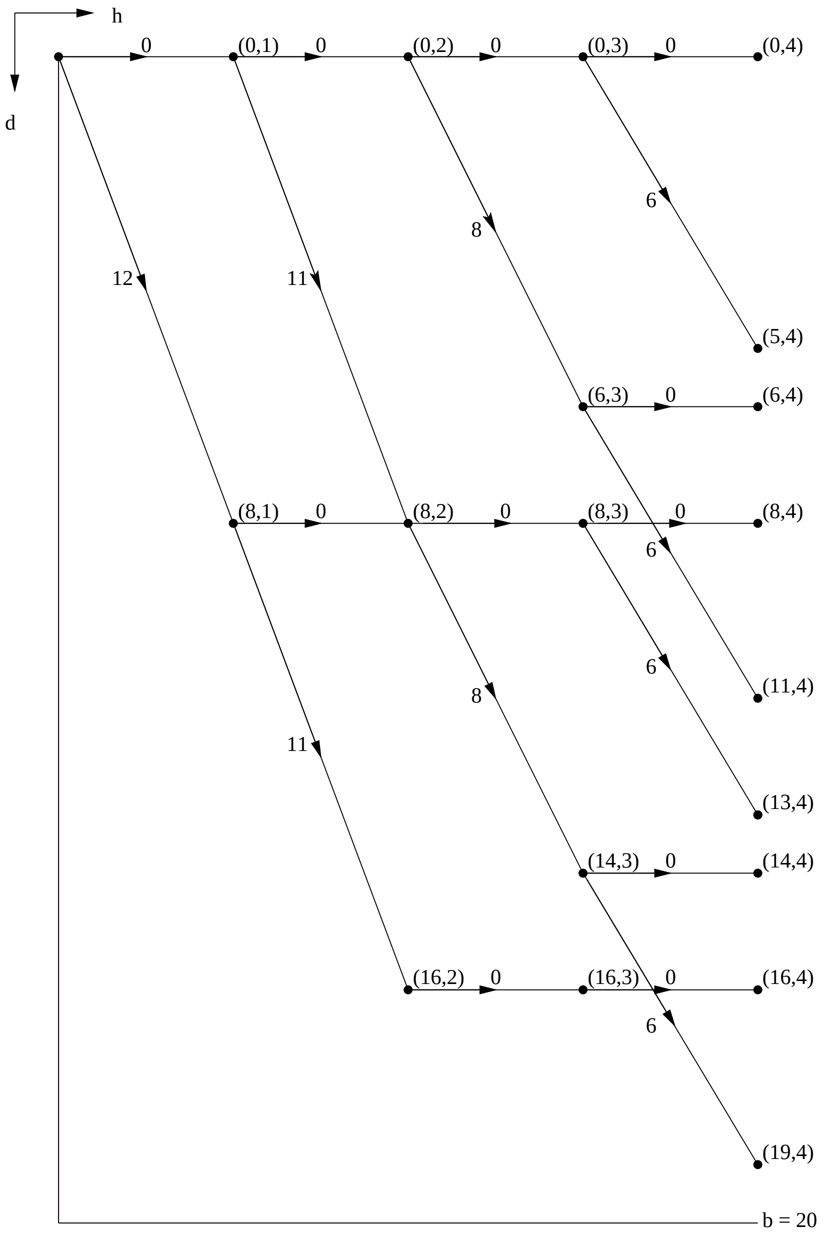
The interpretation is that item k may or may not be part of an optimal filling of the knapsack with capacity d and items $1,\ldots,k$ available. If it is not, then $f_k(d) = f_{k-1}(d)$; if it is, then $f_k(d) = f_{k-1}(d-a_k)+c_k$.

We implement the dynamic programming algorithm as follows. As an initialization, we set $f_0(d) = 0$ for $d = 0,1,\ldots,b$. In iteration k, we add item k and compute $f_k(d)$ for $d = 0,\ldots,b$ on the basis of our knowledge of $f_{k-1}(d)$ for $d = 0,\ldots,b$. The optimal value of the knapsack problem is then equal to $f_n(b)$, and the corresponding solution is determined by tracing the path back. Since $f_k(d)$ is determined in constant time when $f_{k-1}(d)$ and $f_{k-1}(d-a_k)$ are known, we can compute $f_n(b)$ in $O(nb)$ time. This is not polynomial, because the input of the knapsack problem is a linear function of n and $\log b$ (and of $\log a_i$ and $\log c_i$).

We illustrate the dynamic programming algorithm by treating it as a longest path problem with vertices (k,d) $(k = 0,\ldots,n; d = 0,\ldots,b)$. There is an arc of cost 0 from $(k-1,d)$ to (k,d) and an arc of cost c_k from $(k-1,d-a_k)$ to (k,d). For each k, we need only a part of the values: if $f_k(d+1) = f_k(d)$, then it is not necessary to maintain $f_k(d+1)$. This fact is used in the diagram of the example presented below. For illustrative reasons, many arcs are left out of the diagram.

Example 4 items; $b = 20$

item i	1	2	3	4
c_i	12	11	8	6
a_i	8	8	6	5



The optimal value is $f_4(20) = f_4(19) = 26$.

The question which of the two exact algorithms is to be preferred does not have a unique answer. Empirical analysis suggests that problems with many items are solved fastest by branch bound and problems with few items are best solved by the dynamic programming approach. A possible reason for it is that the dynamic programming algorithm grows linearly with n, whereas branch and bound usually grow sublinearly with n, due to the good lower and upper bounding capability; in case of bad luck, however, we may have to inspect all possible subsets, of which there are $O(2^n)$.

Section 4.5. Approximation algorithms

If we do not insist on finding an optimal solution, we can resort to an approximation algorithm for a quickly determined solution, which unfortunately can be quite bad. It is easy to develop an approximation algorithm; usually, it is much more difficult to estimate the quality of the outcome of the algorithm. We describe three approximation algorithms for the knapsack problem and a method of how to measure the quality of an approximation algorithm.

Approximation algorithm 1.
Round down the solution of the continuous knapsack problem. Denote its value by z_1.

Approximation algorithm 2.
Fill the knapsack with item i with the highest value c_i only. Denote its value by z_2.

An example for which Approximation algorithm 1 performs poorly is the following:

$n = 2; \qquad b = 10;$

i	1	2
c_i	2	10
a_i	1	10

Approximation algorithm 1 rounds down the solution $(1, \frac{9}{10})$ to $(1, 0)$ and finds a solution with value $z_1 = 2$, whereas the value $z_{opt} = 10$ is optimal; this yields the relative performance $\frac{z_1}{z_{opt}} = \frac{1}{5}$. This performance can be made arbitrary small in the following way: take the instance $n = 2; c_1 = 2, a_1 = 1, c_2 = a_2 = x; b = x$, where x is a very large integer. Note that Approximation algorithm 2 provides the optimal solution.

An example where Approximation algorithm 2 performs poorly is the following:

$n = 6; \qquad b = 6;$

i	1	2	3	4	5	6
c_i	1	1	1	1	1	1
a_i	1	1	1	1	1	1

Approximation algorithm 2 just picks one item and hence $z_2 = 1$, whereas $z_{opt} = 6$; this yields the relative performance $\frac{z_2}{z_{opt}} = \frac{1}{6}$. This performance can also be made arbitrarily small: take the instance with $n = b = x$ and $a_i = c_i = 1$ for $i = 1, \ldots, x$. Approximation algorithm 1 provides the optimal solution.

The question whether there are instances for which both approximation algorithms behave poorly is answered by examining the third approximation algorithm.

Approximation algorithm 3.
Take the best solution of Approximation algorithms 1 and 2, that is, its value, denoted by z_3, is set equal to $z_3 \leftarrow \max\{z_1, z_2\}$.

Consider the following example:

$n = 3; \qquad b = 10;$

i	1	2	3
c_i	6	5	5
a_i	6	5	5

For the above example, both Approximation algorithm 1 and Approximation algorithm 2 find a solution of value 6, which implies $z_3 = 6$. Since the optimal solution value is 10, we get the relative performance $\frac{z_3}{z_{opt}} = \frac{6}{10}$. It seems that one cannot do much worse. In fact, there is a hard guarantee on the behavior of Approximation algorithm 3 concerning the relative error.

Theorem 4.2.
Approximation algorithm 3 has worst-case bound 1/2, that is, for all instances of knapsack, Approximation algorithm 3 finds a solution with value at least half the value of the optimal solution, and this bound can be approximated arbitrarily closely.

Proof.
Consider any instance of knapsack. Let the continuous solution be $x_1 = \ldots x_{k-1} = 1, x_k = \varepsilon < 1$, $x_{k+1} = \ldots = x_n = 0$; let Approximation algorithm 2 choose item i with $c_i \geq c_k$. Since $z_{opt} \leq z_{CKS} < c_1 + \ldots + c_{k-1} + c_k \leq c_1 + \ldots + c_{k-1} + c_i = z_1 + z_2 \leq 2z_3$, we have that $z_3 > \frac{1}{2} z_{opt}$.

A type of example with which the worst-case bound of 1/2 can be approximated arbitrarily closely is the following one: Take $n = 3$ with $c_1 = a_1 = x + 1$, $c_2 = a_2 = c_3 = a_3 = x$, and $b = 2x$, where x is some large integer; the instance with $x = 5$ is shown above. The performance of Approximation algorithm 3 amounts to $(x + 1)/2x$, which goes to 1/2 if x goes to infinity. □

Section 4.6. Complexity

For a given problem, the first question is to decide whether it is polynomially solvable or not. Proving that a problem is polynomially solvable can be done by providing a polynomial algorithm that solves the problem to optimality. But what if there is no such algorithm available? Either there is an efficient algorithm that we cannot find, or there is no such algorithm. The question is how to prove that there is no polynomial algorithm available. Unfortunately, we do not know. There is a huge class of problems for which no polynomial optimization algorithms are known, but nobody has been able to prove that such algorithms do not exist. The knapsack problem belongs to this class. But for a subclass of these problems, the following nice result is knows: if we can find a polynomial algorithm that solves one of the problems in this class, then we can solve all problems in this subclass in polynomial time. So, for lack of something better, one is left with showing that the problem under consideration belongs to this subclass. This proof proceeds in the following way. We take an arbitrary instance of a problem that is known to belong to this subclass (so we will call it a *hard* problem) and show how to construct a special instance of the problem under consideration *whose outcome is identical to the outcome of the general instance of the hard problem.* The thought behind it is that we can solve the hard problem by solving the problem under consideration: this must then be hard, too.

To show that the knapsack problem is hard too, we use a well-known simply formulated problem: the partition problem.

Partition

Given a set of t weights $u_1, \ldots, u_t$ with $\sum_{i=1}^{t} u_i = 2u$, the question is: Can these weights be partitioned into two sets S and $\{1, \ldots, t\} \backslash S$ of equal cumulative weight, that is, $\sum_{i \in S} u_i = u$?

Researchers have not been able to come up with an efficient algorithm for partition yet. Therefore, it *seems* unlikely that an efficient algorithm for partition exists. To show that the same holds for the knapsack problem, it suffices to show that partition is a special case of the knapsack problem.

Consider any instance of Partition with t weights, $u_1, \ldots, u_t$; $u = \frac{1}{2}\sum_{i=1}^{t} u_i$.

We construct our special instance of the knapsack problem as follows: For each weight u_i in partition, we introduce an item i with both weight and value equal to u_i; the capacity of the knapsack is set equal to u. This instance is depicted below:

$$
\begin{aligned}
&n \leftarrow t,\\
&c_i \leftarrow u_i \; (i = 1, \ldots, n),\\
&a_i \leftarrow u_i \; (i = 1, \ldots, n),\\
&b \leftarrow u.
\end{aligned}
$$

The corresponding instances of the knapsack and the partition problem are equivalent in the following sense.

Theorem 4.3.
An instance $(t, u_1, \ldots, u_t, u)$ of partition has a solution if and only if the corresponding instance of knapsack has an optimal solution with value equal to u.

Proof.
Suppose that the instance of partition has a solution; let S be the corresponding solution. For the corresponding instance of knapsack, we choose $x_i = 1$ if $i \in S$ and $x_i = 0$, otherwise. We have that

$$\sum_{i=1}^{n} a_i x_i = \sum_{i \in S} a_i = \sum_{i \in S} u_i = u = b \text{ , and}$$

$$\sum_{i=1}^{n} c_i x_i = \sum_{i \in S} c_i = \sum_{i \in S} u_i = u \text{ .}$$

This is optimal, since $u = b \geq \sum_{i=1}^{n} a_i x_i = \sum_{i=1}^{n} c_i x_i$.

On the other hand, suppose that the knapsack problem has an optimal solution with value b. Let S be the set of items i for which $x_i = 1$. Then we have that $u = \sum_{i=1}^{n} c_i x_i = \sum_{i \in S} c_i = \sum_{i \in S} u_i$, which implies that partition has a solution. □

We are not completely done yet. In the previous section on dynamic programming, we showed that it is possible to solve a knapsack problem by solving an instance of the longest path problem in an acyclic network; a problem that is known to be solvable in polynomial time. The instance of the longest path problem, however, was much bigger ($O(nb)$) than the instance of the knapsack problem ($O(n \log b)$). We need the extra condition that the size of the knapsack instance corresponding to the instance of partition must be polynomial in the size of the instance of partition. Now that we have checked that this condition is satisfied, we have that knapsack is at least as hard as partition. Note that the knapsack problem does not have to be just as hard as partition; to show that, we must show that the knapsack problem contains partition as a special case, which can easily be shown.