Chapter 4

Knapsack

This chapter is concerned with the Knapsack problem. This problem is of interest in its own right because it formalizes the natural problem of selecting items so that a given budget is not exceeded but profit is as large as possible. Questions like that often also arise as subproblems of other problems. Typical applications include: option-selection in finance, cutting, and packing problems.

In the Knapsack problem we are given a budget W and n items. Each item j comes along with a profit c_j and a weight w_j. We are asked to choose a subset of the items as to maximize total profit but the total weight not exceeding W.

Example 4.1. We are given an amount of W and we wish to buy a subset of n items and sell those later on. Each such item j has cost w_j but yields profit c_j. The goal is to maximize the total profit. Consider $W = 100$ and the following profit-weight table:

j	c_j	w_j
1	150	100
2	2	1
3	55	50
4	100	50

Our choice of purchased items must not exceed our capital W. Thus the feasible solutions are $\{1\}, \{2\}, \{3\}, \{4\}, \{2,3\}, \{2,4\}, \{3,4\}$. Which is the best solution? Evaluating all possibilities yields that $\{3,4\}$ gives 155 altogether which maximizes our profit.

Problem 4.1 Knapsack

Instance. Non-negative integral vectors $c \in \mathbb{N}^n, w \in \mathbb{N}^n$, and an integer W.

Task. Solve the problem

$$\begin{aligned}
\text{maximize} \quad & \text{val}(x) = \sum_{j=1}^{n} c_j x_j, \\
\text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \le W, \\
& x_j \in \{0,1\} \quad j = 1, \dots n.
\end{aligned}$$

For an *item* j the quantity c_j is called its *profit*. The profit of a vector $x \in \{0,1\}^n$ is $\mathrm{val}(x) = \sum_{j=1}^{n} c_j x_j$.

The number w_j is called the *weight* of item j. The *weight* of a vector $x \in \{0,1\}^n$ is given by $\mathrm{weight}(x) = \sum_{j=1}^{n} w_j x_j$. In order to obtain a non-trivial problem we assume $w_j \leq W$ for all $j = 1, \dots, n$ and $\sum_{j=1}^{n} w_j > W$ throughout.

Knapsack is NP-hard which means that "most probably", there is no polynomial time optimization algorithm for it. However, in Section 4.1 we derive a simple 1/2-approximation algorithm. In Section 4.3 we can even improve on this by giving a polynomial-time $1-\varepsilon$-approximation algorithm (for every fixed $\varepsilon > 0$).

4.1 Fractional Knapsack and Greedy

A direct relaxation of Knapsack as an LP is often referred to as the Fractional Knapsack problem:

$$\begin{aligned} \text{maximize} \quad & \mathrm{val}(x) = \sum_{j=1}^{n} c_j x_j, \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq W, \\ & 0 \leq x_j \leq 1 \quad j = 1, \dots, n. \end{aligned}$$

This problem is solvable in polynomial time quite easily. The proof of the observation below is left as an exercise.

Observation 4.2. *Let $c, w \in \mathbb{N}^n$ be non-negative integral vectors with*

$$\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n}$$

and let

$$k = \min \left\{ j \in \{1, \dots, n\} : \sum_{i=1}^{j} w_i > W \right\}.$$

Then an optimum solution for the Fractional Knapsack *problem is given by*

$$\begin{aligned} x_j &= 1 \quad \text{for } j = 1, \dots, k-1, \\ x_j &= \frac{W - \sum_{i=1}^{k-1} w_i}{w_k} \quad \text{for } j = k, \text{ and} \\ x_j &= 0 \quad \text{for } j = k+1, \dots, n. \end{aligned}$$

The ratio c_j/w_j is called the *efficiency* of item j. The item number k, as defined above, is called the *break item*.

Now we turn our attention back to the original Knapsack problem. We may assume that the items are given in non-increasing order of efficiency. Observation 4.2 suggests the following simple algorithm: $x_j = 1$ for $j = 1, \dots, k-1$, $x_j = 0$ for $j = k, \dots, n$.

Unfortunately, the approximation ratio of this algorithm can be arbitrarily bad as the example below shows. The problem is that more efficient items can "block" more profitable ones.

Example 4.3. Consider the following instance, where W is a sufficiently large integer.

j	c_j	w_j	c_j/w_j
1	1	1	1
2	$W-1$	W	$1-1/W$

The algorithm chooses item 1, i.e., the solution $x = (1, 0)$ and hence $\mathrm{val}(x) = 1$. The optimum solution is $x^* = (0, 1)$ and thus $\mathrm{val}(x^*) = W - 1$. The approximation ratio of the algorithm is $1/(W - 1)$, i.e., arbitrarily bad. However, this natural algorithm can be turned into a 1/2-approximation.

Algorithm 4.1 GREEDY

Input. Integer W, vectors $c, w \in \mathbb{N}^n$ with $w_j \le W$, $\sum_j w_j > W$, and $c_1/w_1 \ge \cdots \ge c_n/w_n$.

Output. Vector $x \in \{0, 1\}^n$ such that $\mathrm{weight}(x) \le W$.

```
Step 1. Define $k = \min\{j \in \{1, \dots, n\} : \sum_{i=1}^{j} w_i > W\}$.
Step 2. Let $x$ and $y$ be the following two vectors: $x_j = 1$ for $j = 1, \dots, k-1$, $x_j = 0$ for
        $j = k, \dots, n$, and $y_j = 1$ for $j = k$, $y_j = 0$ for $j \ne k$.
Step 3. If $\mathrm{val}(x) \ge \mathrm{val}(y)$ return $x$ otherwise return $y$.
```

Theorem 4.4. *The algorithm* GREEDY *is a* 1/2*-approximation for* KNAPSACK.

Proof. The value obtained by the GREEDY algorithm is equal to $\max\{\mathrm{val}(x), \mathrm{val}(y)\}$.

Let x^* be an optimum solution for the KNAPSACK instance. Since every solution that is feasible for the KNAPSACK instance is also feasible for the respective FRACTIONAL KNAPSACK instance we have that

$$\mathrm{val}(x^*) \le \mathrm{val}(z^*),$$

where z^* is the respective optimum solution for FRACTIONAL KNAPSACK. Observe that it has the structure $z^* = (1, \dots, 1, \alpha, 0, \dots, 0)$, where $\alpha \in [0, 1)$ is at the break item k. The solutions x and y are $x = (1, \dots, 1, 0, 0, \dots, 0)$ and $y = (0, \dots, 0, 1, 0, \dots, 0)$.

In total we have

$$\mathrm{val}(x^*) \le \mathrm{val}(z^*) = \mathrm{val}(x) + \alpha c_k \le \mathrm{val}(x) + \mathrm{val}(y) \le 2 \max\{\mathrm{val}(x), \mathrm{val}(y)\}$$

which implies the approximation ratio of 1/2. □

4.2 Pseudo-Polynomial Time Algorithm

Here we give a pseudo-polynomial time algorithm that solves KNAPSACK optimally by using dynamic programming. The term *pseudo-polynomial* means polynomial if the input is given in unary encoding (and thus exponential if the input is given in binary encoding).

The idea is the following: Suppose you restrict yourself to choose only among the first j items, for some integer $j \in \{0, \dots, n\}$. So all the solutions x you consider have the form $x_i \in \{0, 1\}$ for $i = 1, \dots, j$ and $x_i = 0$ for $i = j + 1, \dots, n$. With abuse of notation write $x \in \{0,1\}^j 0^{n-j}$. Now the variable $m_{j,k}$ equals the minimum total weight of such a solution x with $\text{weight}(x) \leq W$ and $\text{val}(x) = k$. That is, after defining the set $W_{j,k} = \{\text{weight}(x) : \text{weight}(x) \leq W, \text{val}(x) = k, x \in \{0,1\}^j 0^{n-j}\}$ we require

$$m_{j,k} = \inf W_{j,k}.$$

(Recall that for any finite set S of integers $\inf S = \min S$ if $S \neq \emptyset$ and $\inf S = \infty$, otherwise.)

Let C be any upper bound on the optimum profit, for example $C = \sum_i c_i$. Clearly, the value of an optimum solution for Knapsack is the largest value $k \in \{0, \ldots, C\}$ such that $m_{n,k} < \infty$. The algorithm Dynamic Programming Knapsack recursively computes the values for $m_{j,k}$ and then returns the optimum value for the given Knapsack instance. In the algorithm below, the variables $x(j,k)$ are n-dimensional vectors that store the solutions corresponding to $m_{j,k}$, i.e., with weight equal to $m_{j,k}$ and value k.

Algorithm 4.2 Dynamic Programming Knapsack

```
Input.   Integers $W, C$, vectors $w, c \in \mathbb{N}^n$.

Output.  Vector $x \in \{0,1\}^n$ such that $\text{weight}(x) \leq W$.

Step 1. Set $m_{0,0} = 0$, $m_{0,k} = \infty$ for $k = 1, \ldots, C$, and $x(0,0) = 0$.

Step 2. For $j = 1, \ldots, n$ and $k = 0, \ldots, C$ do

        $m_{j,k} = \begin{cases} m_{j-1,k-c_j} + w_j & \text{if } c_j \leq k \text{ and } m_{j-1,k-c_j} + w_j \leq \min\{W, m_{j-1,k}\}, \\ m_{j-1,k} & \text{otherwise.} \end{cases}$

        If the first case applied set $x(j,k)_i = x(j-1,k-c_j)_i$ for $i \neq j$ and $x(j,k)_j = 1$.
        Otherwise set $x(j,k) = x(j-1,k)$.

Step 3. Determine the largest $k \in \{0, \ldots, C\}$ such that $m_{n,k} < \infty$. Return $x(n,k)$.
```

Theorem 4.5. *The* Dynamic Programming Knapsack *algorithm computes the optimum value of the* Knapsack *instance W, $w, c \in \mathbb{N}^n$ in time $O(nC)$, where C is an arbitrary upper bound on this optimum value.*

Proof. The running time is obvious. For the correctness we prove that the values $m_{j,k}$ computed by the algorithm satisfy

$$m_{j,k} = \inf W_{j,k}$$

by induction on j. Here $W_{j,k} = \{\text{weight}(x) : \text{weight}(x) \leq W, \text{val}(x) = k, x \in \{0,1\}^j 0^{n-j}\}$ by definition.

The base case $j = 0$ is clear. For the inductive case first consider a situation when the algorithm sets

$$m_{j,k} = m_{j-1,k-c_j} + w_j,$$

i.e. we "take" the j-th item. Let $y = x(j-1, k-c_j)$ be the solution that corresponds to $m_{j-1,k-c_j}$. The solution $x = x(j,k)$ that corresponds to $m_{j,k}$ is obtained from y by setting $x_i = y_i$ for $i \neq j$ and $x_j = 1$. The value of x is $\text{val}(x) = k$. By definition of the algorithm we have $\text{weight}(x) = \text{weight}(y) + w_j = m_{j-1,k-c_j} + w_j \leq W$ and thus $x \in W_{j,k}$.

By construction of the algorithm and induction hypothesis we have $\text{weight}(x) \le \inf W_{j-1,k}$ and $\text{weight}(x) = w_j + \inf W_{j-1,k-c_j}$. That is, the weight of x is at most the weight of any solution without the j-th item and at most the weight of any solution including the j-th item. Hence $m_{j,k} = \inf W_{j,k}$.

In the other situation, when the algorithm sets

$$m_{j,k} = m_{j-1,k},$$

then either $c_j > k$ and hence no solution with value equal to k can contain the j-th item, or $m_{j-1,k} + w_j > W$, i.e., adding the j-th item is infeasible, or $m_{j-1,k} + w_j > \inf W_{j-1,k}$, i.e., there is a solution with less weight and still value equal to k. □

4.3 Fully Polynomial-Time Approximation Scheme

Here we give a *fully polynomial time approximation scheme* (FPTAS), i.e., we show that for every fixed $\varepsilon > 0$ there is an $1 - \varepsilon$-approximation algorithm that runs in time polynomial in the input size and $1/\varepsilon$. From a complexity-theoretic point of view this is the best that can be hoped for: Assuming $\mathsf{P} \neq \mathsf{NP}$ there is no polynomial time algorithm that solves Knapsack optimally on every instance, but the FPTAS delivers solutions with arbitrarily good approximation guarantees in polynomial time. (Unfortunately not many problems admit an FPTAS.)

A common theme in constructing FPTASs is the following: First find an algorithm that solves the problem exactly (mostly using the dynamic programming paradigm). This algorithm usually has pseudo-polynomial or even exponential running time. Second construct an algorithm for "rounding" input-instances, i.e., reducing the input-size. This modification reduces the running time but may lead to inaccurate solutions.

The running time of Dynamic Programming Knapsack is $O(nC)$. If we divide we profit c_j of each item by a number t and round the result down, then this improves the running time of Dynamic Programming Knapsack by a factor of t to $O(nC/t)$ but may yield suboptimal solutions.

Algorithm 4.3 Knapsack FPTAS

Input. Integer W, vectors $w, c \in \mathbb{N}^n$, a number $\varepsilon > 0$.

Output. Vector $x \in \{0,1\}^n$ such that $\text{weight}(x) \le W$.

Step 1. Run Greedy on the instance W, w, c and let x be the solution. If $\text{val}(x) = 0$ then return x.

Step 2. Set $t = \max\{1, \varepsilon \text{val}(x)/n\}$ and set

$$c'_j = \left\lfloor \frac{c_j}{t} \right\rfloor \quad \text{for } j = 1, \ldots, n.$$

Step 3. Set $C = 2\text{val}(x)/t$ and apply the Dynamic Programming Knapsack algorithm on the instance W, C, w, c' and let y be the solution obtained.

Step 4. If $\text{val}(x) \ge \text{val}(y)$ return x otherwise y.

Theorem 4.6. *For every fixed $\varepsilon > 0$, the* Knapsack FPTAS *algorithm is a $1-\varepsilon$-approximation algorithm with running time $O\left(n^2/\varepsilon\right)$.*

Proof. The value of the solution returned by the algorithm is equal to $\max\{\mathrm{val}(x), \mathrm{val}(y)\}$. Let x^* be an optimum solution for the instance W, w, c. By Theorem 4.4 we have $2\mathrm{val}(x) \geq \mathrm{val}(x^*)$ and hence the choice $C = 2\mathrm{val}(x)/t$ is a legal upper bound for the optimum value of the rounded instance W, w, c'. By Theorem 4.5 y is an optimum solution for this instance and we have

$$\begin{aligned}\mathrm{val}(y) &= \sum_{j=1}^{n} c_j y_j \geq \sum_{j=1}^{n} t c'_j y_j = t \sum_{j=1}^{n} c'_j y_j \\ &\geq t \sum_{j=1}^{n} c'_j x^*_j = \sum_{j=1}^{n} t c'_j x^*_j > \sum_{j=1}^{n} (c_j - t) x^*_j \geq \mathrm{val}(x^*) - nt.\end{aligned}$$

If $t = 1$ then y is optimal by Theorem 4.5. Otherwise the above inequality and the choice of t yields $\mathrm{val}(y) \geq \mathrm{val}(x^*) - \varepsilon\mathrm{val}(x)$ and hence

$$\mathrm{val}(x^*) \leq \mathrm{val}(y) + \varepsilon\mathrm{val}(x) \leq (1+\varepsilon)\max\{\mathrm{val}(x), \mathrm{val}(y)\}$$

which yields the approximation guarantee $1 - \varepsilon/(1+\varepsilon)$.

The running time of Dynamic Programming Knapsack on the rounded instance is

$$O\left(nC\right) = O\left(\frac{n\mathrm{val}(x)}{t}\right) = O\left(\frac{n^2}{\varepsilon}\right),$$

where we have used the definition of t: If $t = 1$ then $\mathrm{val}(x) \leq n/\varepsilon$ and otherwise $t = \varepsilon\mathrm{val}(x)/n$. This running time dominates the time needed for the other steps. □