Moderne Suchmethoden der Informatik

The knapsack problem

Florian Pawlik

December 1, 2014

Abstract

Have you ever wondered about how heavy one can fill his knapsack? No? That does not matter. You will get the answer in this paper. There are several algorithms for the simple knapsack problem and the knapsack problem. Four of them will be shown and proved in this paper.

Contents

1 Introduction **3**

2 The problems **3**
2.1 The knapsack problem 3
2.2 The simple knapsack problem 3

3 Algorithms for SKP **3**
3.1 Greedy algorithm 4
3.2 Polynomial time approximation scheme 4
3.3 Modifying the SKP 6
3.4 PTAS for modified SKP 7

4 Algorithms for KP **8**
4.1 Algorithm using dynamic programming 8
4.2 Fully polynomial time approximation scheme 8

1 Introduction

The knapsack problem(KP)(see Section 2.1) is in the class of the NP hard problems and one of the most famous problems. Before we find an algorithm for the KP, we search for some algorithms(see Section 3) for the simple knapsack problem(SKP)(see Section 2.2) such as the greedy algorithm(see Section 3.1). Then we find a PTAS(see Section 3.2) for the SKP. After that we modify our SKP by introducing a second parameter for each index and defining the DIST-function(see Section 3.3). We then find a PTAS for the modified SKP(see Section 3.4). The last section is about the KP and a FPTAS(see Section 4.2) to solve it. Within the FPTAS we will see an algorithm(see Section 4.1) to solve KP correctly using dynamic programming.

2 The problems

The two Problems we are facing in this paper can easily be defined as a maximisation problem. They can be formulated in mathematical language as follows.

2.1 The knapsack problem

The main problem to solve in this paper is the knapsack problem. Imagine you have a knapsack that can hold x kilogram and wish to pack several items into it. Each item has a certain value and a weight. The problem is to maximise the cost[1] of your knapsack.

In mathematical language:
maximise $cost(T)$ where $T \subset I$ with $I = \{1, ..., n\}$ is a finite set of indices and
$cost(T) := \sum_{i \in T} c_i$
with the restriction that $\sum_{i \in T} w_i < x$
where c_i means the cost of the item with the index i
and w_i means the weight of the item with the index i.

2.2 The simple knapsack problem

For reasons of simplicity we first use the simple knapsack problem. It is similar to the KP(see Section 2.1), the only difference between them is that in SKP there is $w_i = c_i \forall i \in I$.

To have a picture of what SKP means you take a thief robbing a bank. He has a knapsack that holds x kilogram of weight. There are gold ingots and coins with different weight in the bank. The only value is the gold itself. His goal is to maximise the weight and so the value of his items in the knapsack.

In mathematical language:
maximise $cost(T)$ where $T \subset I$ with $I = \{1, ..., n\}$ is a finite set of indices and
$cost(T) := \sum_{i \in T} w_i$
with the restriction that $\sum_{i \in T} w_i < x$
where w_i means the weight and the cost of the item with the index i.

3 Algorithms for SKP

The important point of solving a problem is to face it with an algorithm. The simplest way of getting a solution is a greedy way. The runtime of a greedy algorithm(see Section 3.1) is good, but the approximation ratio is mostly very bad. That‘s why we search for PTAS(see Section 3.2)(see Section 3.4) algorithms.

[1] cost is just the general notation of the price the single items have in the knapsack. In this example it has the same meaning like value or the price

3.1 Greedy algorithm

The main idea of a greedy algorithm is to do the best thing, that can be done at this point of time. It is not the best solution. We will see, that the greedy algorithm for the SKP has an approximation ratio of 2. The algorithm works according to the following rules: take the index with the highest weight you have not used jet. Then try to add it to your set of indices, therefore you have to look if the sum of the weights of your indices + the new index is heavier than the maximum weight of the knapsack. Add the item if it is possible.

In pseudo code the algorithm looks like:

input: positive integer $w_1, w_2, ..., w_n, b$ for a n$\in \mathbb{N}$.
output: T
step 1: sort w_1 to w_n descending on the size
step 2: $T := \emptyset$; $cost_T := 0$
step 3: **for** i=1 **to** n **do**
if $cost_T + w_i < b$ **then**
do begin $T := T \cup \{i\}$;
$cost_T := cost_T + w_i$;
end

where w_i mean the weight of the index i.
b is the maximum of the sum of the weights a set of indices is allowed to contain. After writing an algorithm we have to take a look on the runtime and the approximation ratio.

The runtime of the greedy algorithm is $O(n \cdot log(n))$ because of step 1.
Sorting in step 1 needs a time of $O(n \cdot log(n))$.
Step 2 runs in a constant time $O(1)$
Step 3 runs in a time depending on the number of iterations of the for-block. The for-block itself runs in a constant time. We get $O(n * O(1)) = O(n)$.

To show that the greedy algorithm is a 2-approximation algorithm for the SKP we have to show that $cost_T \geq \frac{b}{2}$ or T is optimal.

w.l.o.g. we can say that $b \geq w_1 \geq w_2 \geq ... \geq w_n$.
Let $j+1$ be the smallest index not in T.
If we have $j = 1$ then $w_1 + w_2 \geq b$ and $1 \in T$ so $cost_T \geq w_1$. Because of $w_1 \geq w_2$ it is obvious, that $w_1 > \frac{b}{2}$. We get $cost_T \geq w_1 \geq \frac{b}{2}$ and we are finished with the case of $j = 1$.

Lets have a look on $j \geq 2$:
T contains the j smallest indices. In other words the j highest weights are in the knapsack and if you want to put $j+1$ into it it is bigger than b. You can make following approximation for the $j+1$th weight:

$$w_{j+1} \leq w_j \leq \frac{w_1 + w_2 + ... + w_j}{j} \leq \frac{b}{j}$$

Now follows for the $cost_T$ that $cost_T > b - w_{j+1} \geq b - \frac{b}{j} \geq \frac{b}{2}$ for $j \geq 2$
With this result you see, that the Algorithm is a 2-Approximation.

3.2 Polynomial time approximation scheme

According to the inequation(3.1) we can create a PTAS. If a found knapsack T contained the j highest weights out of the optimal knapsack T_{OPT}, then the difference would be $cost_{T_{OPT}} - cost_T \leq w_{i_{j+1}} \leq \frac{b}{i_{j+1}-1} \leq \frac{b}{j}$.

We now search for the j highest weights in T_{OPT} and fill the remaining knapsack using the step 3 of our greedy algorithm(see Section 3.1).

In pseudo code our PTAS looks like

Algorithm 2

input: positive integer $w_1, w_2, ..., w_n, b$ for a n$\in \mathbb{N}$, $\epsilon \in \mathbb{R}$ $0 < \epsilon < 1$.

step 1: sort w_1 to w_n descending on their size

step 2: $j := \lceil \frac{1}{\epsilon} \rceil$.

step 3: for every subset $T \subseteq \{1, 2, ..., n\}$ with $|T| \leq j$ and $\sum_{i \in T} w_i \leq b$, expand T to T^* using step 3 of the greedy algorithm (see Section 3.1).
save the T^* with the highest $cost_{T^*}$.

output: T^* with the maximum of $cost_{T^*}$ out of all T created in the algorithm

To show that algorithm 2 is a PTAS, we need to have a look at the runtime and the approximation rate of the algorithm.

First look at the runtime, which should be polynomial in n for a fixed ϵ.

Step 1 runs in $O(n \cdot log(n))$

Step 2 runs in $O(1)$

Step 3 has to run step 3 of algorithm 1 for every subset S with $|S| \leq k$. Step 3 of algorithm 1 runs in $O(n)$. There are $\sum\limits_{0 \leq i \leq j} \frac{n}{i} \leq \sum\limits_{0 \leq i \leq j} n^i = \frac{n^{j+1}-1}{n-1} = O(n^j)$ subsets to build. Each of them can be built in time $O(1)$, because you can construct one subset out of the subset before using a lexicographical order.

Therefore step 3 runs in $O(n^j) \cdot O(1) \cdot O(n) = O(n^{j+1})$.

The Time of the algorithm is in $max(O(n \cdot log(n)), O(1), O(n^{j+1})) = O(n^{j+1}) = O(n^{\lceil \frac{1}{\epsilon} \rceil})$. This is a polynomial runtime for fixed ϵ.

Now we have to show the approximation rate is under $1 + \epsilon$.

$T_{OPT} = \{i_1, i_2, ..., i_p\}, i_1 \leq i_2 \leq ... \leq i_p$ is an optimal solution for our instance.

If $p \leq j$ we found the optimal solution while searching all the subsets S with $|S| \leq j$. So we are finished here.

If $p > j$ the algorithm 2 step 3 creates a $T = \{i_1, i_2, ..., i_j\}$ which are the indices of the j highest weights of T_{OPT}. If $T^* = T_{OPT}$ we are finished again.

Let be $T^* \neq T_{OPT}$. There is a $i_q \in T_{OPT} - T*$ such that i_q is the smallest index in T_{OPT} and not in T^*. It is notorious that $i_q > i_j \geq j$ and $cost_{T^*} + w_{i_q} > b \geq cost_{T_{OPT}}$.

Furthermore we have $w_{i_q} \leq \frac{w_{i_1} + w_{i_2} + ... + w_{i_j} + w_{i_q}}{j+1} \leq \frac{cost_{T_{OPT}}}{j+1}$.

We then get for the approximation ratio $R(I, \epsilon) =$

$= \frac{cost_{T_{OPT}}}{cost_{T^*}} \leq$

$\leq \frac{cost_{T_{OPT}}}{cost_{T_{OPT}} - w_{i_q}} \leq$

$\leq \frac{cost_{T_{OPT}}}{cost_{T_{OPT}} - (cost_{T_{OPT}}/j+1)} =$

$= \frac{1}{1-(1/j+1)} =$

$= \frac{j+1}{j} =$

$= 1 + \frac{1}{j} \leq$

$\leq 1 + \epsilon$

To get to the knapsack problem we need to introduce the c_i representing the costs of the i-th index with weight w_i. We modify our SKP using a function DIST we will introduce now.

3.3 Modifying the SKP

The DIST-function on input $I = (w_1, ..., w_n, b, c_1, ..., c_n)$ represents the relative distance between w_i and c_i.

$$DIST(I) = max\{max\{\frac{c_i - w_i}{w_i} | c_i \geq w_i, i \in \{1, ..., n\}\}, max\{\frac{w_i - c_i}{c_i} | w_i \geq c_i, i \in \{1, ..., n\}\}\}.$$

$cost_T$ now is the sum of all c_i where $i \in T$.

For the next Algorithm we work on the KP_δ. KP_δ is the subset of all knapsack problems, where $DIST(I) \leq \delta$. Thus we now have $c_i \leq (1+\delta)w_i$ and $w_i \leq (1+\delta)c_i$.

We notate all $(1+\epsilon)$-approximation algorithms built according to the pseudo code of algorithm 2 $\{ASKP_\epsilon\}_{\epsilon>0}$.

We want to show, that for our new problem KP_δ each algorithm out of $\{AKSP_\epsilon\}$ is an $(1+\epsilon+\delta(2+\delta)\cdot(1+\epsilon))$-approximation algorithm.
First of all let the input be sorted. This can be without loss of generality. So $w_1 \geq w_2 \geq ... \geq w_n$ for our input $I = w_1, ..., w_n, b, c_1, ..., c_n$.
We declare $j := \lceil \frac{1}{\epsilon} \rceil$.
And have the $T_{OPT} = \{i_1, ..., i_l\} \subseteq \{1, 2, ..., n\}$ as an optimal solution for I.

If l is less or equal to j our $ASKP_\epsilon$ has found an optimal solution for I with $cost_{T_{OPT}}$.

Let l be greater than k. Then $ASKP_\epsilon$ has found an greedy extension of $T = \{i_1, i_2, ..., i_k\}$ to $T^* = \{i_1, i_2, ..., i_j, k_{j+1}, ..., k_{j+r}\}$.
The next step is to show that $cost_U - cost_{T^*}$ is small relative to $cost_U$.
If we look at the sum of weights in T_{OPT} and T^* we have two possibilities. We start with $\sum_{i \in T_{OPT}} w_i - \sum_{k \in T^*} w_k \leq 0$ such that the knapsack packed with our algorithm is heavier than the optimal knapsack.
Because of $\forall i : (1+\delta)^{-1} \leq \frac{c_i}{w_i} \leq 1+\delta$ [2] it is obvious that $cost_{T_{OPT}} = \sum_{i \in T_{OPT}} \leq (1+\delta) * \sum_{i \in T_{OPT}} w_i$ and $cost_{T^*} = \sum_{i \in T^*} \geq (1+\delta)^{-1} * \sum_{i \in T^*} w_i$.
We now get the inequation
$cost_{T_{OPT}} - cost_{T^*}$
$\leq (1+\delta) \cdot \sum_{i \in T_{OPT}} w_i - (1+\delta)^{-1} \cdot \sum_{k \in T^*} w_k$
$\leq (1+\delta) \cdot \sum_{i \in T_{OPT}} w_i - (1+\delta)^{-1} \cdot \sum_{i \in T_{OPT}} w_i$
$= \frac{\delta \cdot (2+\delta)}{1+\delta} \cdot \sum_{i \in T_{OPT}} w_i$
$\leq \frac{\delta \cdot (2+\delta)}{1+\delta} \sum_{i \in T_{OPT}} (1+\delta) c_i$
$= \delta \cdot (2+\delta) \cdot \sum_{i \in T_{OPT}} c_i$
$= \delta \cdot (2+\delta) \cdot cost_{T_{OPT}}$

So we get for the relative difference $\frac{cost_{T_{OPT}} - cost_{T^*}}{cost_{T_{OPT}}} \leq \frac{\delta \cdot (2+\delta) \cdot cost_{T_{OPT}}}{cost_{T_{OPT}}} = \delta \cdot (2+\delta)$.

Now we take a look on the other opportunity and set $d = \sum_{i \in T_{OPT}} w_i - \sum_{k \in T^*} w_k > 0$. We also want to be c the costs of the part of T_{OPT} which has the weights $\sum_{k \in T^*} w_k$.

From the first part we know that $\frac{c - cost_{T^*}}{c} \leq \delta * (2+\delta)$.

[2] it was already mentioned in the introduction of KP_δ

It is also obvious that $d < b - \sum_{k \in T*} w_k \leq w_{i_r}$ for an $r > j$ and $i_r \in U$. We get $d \leq w_{i_r} \leq \frac{w_{i_1}+w_{i_2}+...+w_{i_r}}{r} \leq \frac{\sum_{i \in T_{OPT}} w_i}{k+1} \leq \epsilon \cdot \sum_{i \in T_{OPT}} w_i$.

With the fact that $cost_{T_{OPT}} \leq c + d \cdot (1+\delta)$ we get

$\frac{cost_{T_{OPT}} - cost_{T^*}}{cost_{T_{OPT}}} \leq \frac{c+d\cdot(1+\delta)-cost_{T^*}}{cost_{T_{OPT}}}$

$\leq \frac{c-cost_{T^*}}{cost_{T_{OPT}}} + \frac{(1+\delta)\cdot\epsilon\cdot\sum_{i\in T_{OPT}} w_i}{cost_{T_{OPT}}}$

$\leq \delta \cdot (2+\delta) + (1+\delta)\cdot\epsilon\cdot(1+\delta) = 2\delta + \delta^2 + \epsilon\cdot(1+\delta)^2 = \epsilon + \delta\cdot(2+\delta)\cdot(1+\epsilon)$

3.4 PTAS for modified SKP

The algorithm 2 is stable according to the $DIST$-function, but not superstable. So if you have a fixed delta your solution can get very worse because of the factor $\delta \cdot (2+\delta)$ in the addition. We improve our algorithm by sorting in other way. Our new PTAS sorts according to the cost per weight and not according to the weights.

The pseudo code of our algorithm 3 looks like:

input: positive integers $w_1, w_2, ..., w_n, b, c_1, ..., c_n$ for a certain $n \in \mathbb{N}$, $\epsilon \in \mathbb{R}$ with $0 < \epsilon < 1$.

step 1: sort $\frac{c_1}{w_1}, \frac{c_2}{w_2}, ..., \frac{c_n}{w_n}$, such that $\frac{c_i}{w_i} \geq \frac{c_{i+1}}{w_{i+1}}$ for $i = 1, ..., n-1$.

step 2: $j := \lceil \frac{1}{\epsilon} \rceil$.

step 3: (similar to step 3 in algorithm 2 but with an other order)
for every subset $T \subseteq \{1, 2, ..., n\}$ with $|T| \leq j$ and $\sum_{i \in S} w_i \leq b$, expand T to T^* running step 3 of algorithm 1 3.1.
save the most valueable T^*.

output: the best T^* build in step 3.

This algorithm has an $1 + \epsilon \cdot (1+\delta)^2$-approximation rate for a KP_δ.

To show the rate we have $T_{OPT} = \{i_1, ..., i_l\} \subset \{1, ..., n\}$ as an optimal solution.
If $l \leq j$ the algorithm finds an optimal solution.
If we have $l > j$ we have a expansion T^* of $T = \{i_1, i_2, ..., i_j\}$. To go further we take a look at the sum of weights again. If the sum of weights in T_{OPT} would be less than the sum of weights in T^*, then T^* would have a higher prize because of the order you add the indices. So we can mark this case as not possible.

Let be $d = \sum_{i \in T_{OPT}} w_i - \sum_{j \in T^*} w_j \geq 0$. Let c be the costs of T_{OPT} with the weight $\sum_{j \in T^*} w_j$. Because of the optimality of T^* according to costs per weight it is obvious that $c - cost_{T^*} \leq 0$. The indices $i_1, i_2, ..., i_j$ are in T_{OPT} and in T^* so the weights $w_{i_1}, ..., w_{i_j}$ are the highest weights in both of them. For the rest of weights of T_{OPT} we have:$d \leq \epsilon * \sum_{i \in T_{OPT}} w_i$. We also know that $cost_{T_{OPT}} \leq c + d * (1+\delta)$.
With this information we can calculate:

$\frac{cost_{T_{OPT}} - cost_{T^*}}{cost_{T_{OPT}}}$

$\leq \frac{c+d\cdot(1+\delta)-cost_{T^*}}{cost_{T_{OPT}}}$

$\leq \frac{d\cdot(1+\delta)}{cost_{T_{OPT}}}$

$\leq \epsilon \cdot (1+\delta) \cdot \frac{\sum_{i\in T_{OPT}} w_i}{cost_{T_{OPT}}}$

$\leq \epsilon \cdot (1+\delta)^2$

We now can see that algorithm 3 is a PTAS. The runtime is the same as before in algorithm 2 and we have an approximation linked to epsilon. We can get a solution as good as we want it by choosing the ϵ and so the j in the right size.

4 Algorithms for KP

We have already seen algorithm 3 can also be used on the KP_δ but to get an algorithm for all KP we have to use some other methods. First we find an algorithm(see Section 4.1) using dynamic programming with an exponential time that gets the best solution. After that we will find an FPTAS(see Section 4.2) using the dynamic algorithm with a logarithmic subset of cases. So we get a polynomial runtime.

4.1 Algorithm using dynamic programming

We will not have a long view on this algorithm. It can be looked up in many different books and papers.

Algorithm 4

input: positive integer $w_1, w_2, ..., w_n, b, c_1, ..., c_n$ for a certain $n \in \mathbb{N}$, $\epsilon \in \mathbb{R}^+$.

step 1: $TRIPLE(1) := \{(0, 0, \emptyset)\} \cup \{(c_1, w_1, \{1\}) | if\ w_1 \leq b)\}$.

step 2: **for** $i = 1$ **to** $n - 1$**do**
begin $Set(i + 1) := TRIPLE(i)$
for every $(kw, T) \in TRIPLE(i)$ **do**
if $w + w_{i+1} \leq b$ **then**
$SET(i + 1) := SET(i + 1) \cup \{(k + c_{i+1}, w + w_{i+1}, T \cup \{i + 1\})\}$
SET(i+1) contains just one triple (k,w,T) for each k. This triple has the smallest weight w.

schritt 3: search for the highest k.

output: T

We will see the time complexity in the proof of algorithm 5 being an FPTAS.

4.2 Fully polynomial time approximation scheme

The idea of our FPTAS is to create an input with a logarithmic factor smaller cost and then search for the optimal output for our new and smaller problem. For this we use the dynamic algorithm. This way, we get a polynomial runtime and our algorithm has a good approximation ratio.

algorithm 5

input: positive integer $w_1, w_2, ..., w_n, b, c_1, ..., c_n$ for a certain $n \in \mathbb{N}$, $\epsilon \in \mathbb{R}^+$.

step 1: $c_{max} := max\{c_1, ..., c_n\}$.
$t := \left\lfloor \log_2 \frac{\epsilon \cdot c_{max}}{(1+\epsilon) \cdot n} \right\rfloor$.

step 2: for all $i \in \{1, ..., n\} : c'_i := \left\lfloor c_i \cdot 2^{-t} \right\rfloor$.

step 3: calculate the optimal T' for the input $I' = w_1, ..., w_n, b, c'_1, ..., c'_n$ using algorithm 4.

output: $T = T'$.

To show that this really is an FPTAS we need an approximation ratio of $(1+\epsilon)$.
We know, that T' is an optimal solution for I' and so it is a suitable solution for I'. I and I' do not differ in their weights. That's why T' is a possible solution for I. Let T_{OPT} be an optimal solution for I.

We have $cost(T_{OPT}, I) = \sum_{j \in T_{OPT}} c_j \geq \sum_{j \in T'} c_j = cost(T', I)$
$\geq 2^t \cdot \sum_{j \in T'} c'_j$
$\geq 2^t \cdot \sum_{j \in T_{OPT}} c'_j$
$= \sum_{j \in T_{OPT}} 2^t \cdot \lfloor c_j \cdot 2^{-t} \rfloor$
$\geq \sum_{j \in T_{OPT}} 2^t (c_j \cdot 2^{-t} - 1)$
$\geq (\sum_{j \in T_{OPT}} c_j) - n \cdot 2^t$
$= cost(T_{OPT}, I) - n \cdot 2^t$

We can easily see that $cost(T_{OPT}, I) \geq cost(T', I) \geq cost(T, I - n \cdot 2^t$. Transforming the term to $0 \leq cost(T, I) - cost(T', I) \leq n \cdot 2^t$
With the definition of t we can see, that its smaller than $n \cdot \frac{\epsilon * c_{max}}{(1+\epsilon) \cdot n} = \epsilon \cdot \frac{c_{max}}{1+\epsilon}$. With the assumption of $cost(T, I) \geq c_{max}$ we get $cost(T', I) \geq c_{max} - \epsilon \cdot \frac{c_{max}}{1+\epsilon}$. We can make this assumption because if it would be smaller than c_{max} the weight of the index of c_{max} would not fit into the knapsack, so we can kick of this index and his weight an cost.

With this result we get for our approximation ratio $R(I) = \frac{cost(T_{OPT}, I)}{cost(T', I)}$
$= \frac{cost(T', I) + cost(T_{OPT}, I) - cost(T', I)}{cost(T', I)}$
$\leq 1 + \frac{\epsilon \cdot (c_{max}/(1+\epsilon))}{cost(T', I)}$
$\leq 1 + \frac{\epsilon \cdot (c_{max}/(1+\epsilon))}{c_{max} - \epsilon \cdot \epsilon \cdot (c_{max}/(1+\epsilon))}$
$= 1 + \frac{\epsilon}{1+\epsilon} \cdot \frac{1}{1 - (\epsilon/(1+\epsilon))}$
$= 1 + \frac{\epsilon}{1+\epsilon} \cdot (1+\epsilon) = 1 + \epsilon$

When having a look at the time complexity we can see that step 1 and step 2 run in time $O(n)$.
Step 3 needs the time of $O(n \cdot Opt_{KP}(I'))$ where $Opt_{KP}(I')$ is the time algorithm 4 needs to run. We can bound this time to
$Opt_{KP}(I')$
$\leq \sum_{i=1}^{n} c'_i$
$= \sum_{i=1}^{n} \left\lfloor c_i \cdot 2^{-\left\lfloor \log_2 \frac{\epsilon \cdot c_{max}}{(1+\epsilon) \cdot n} \right\rfloor} \right\rfloor$
$\leq \sum_{i=1}^{n} (c_i \cdot 2 \cdot \frac{(1+\epsilon) \cdot n}{\epsilon \cdot c_{max}})$
$= 2 \cdot (1+\epsilon) \cdot \epsilon^{-1} \cdot \frac{n}{c_{max}} \cdot \sum_{i=1}^{n} c_i$
$\leq 2 \cdot (1+\epsilon) \cdot \epsilon^{-1} \cdot n^2 \in O(\epsilon^{-1} \cdot n^2)$

As you all can see the algorithm runs in time $O(\epsilon^{-1} \cdot n^3)$ and so is a FPTAS.

References

J. Hromkovič: Algorithmics for Hard Problems - Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics, pp. 238-248 and algorithm 3.2.2.2 Springer-Verlag: Berlin-Heidelberg-New York, 2001

Manuel Schmitt: Great oral explanations during the seminar to understand dynamic programming and the time complexity of the FPTAS.