Chapter 3

Approximation Algorithms

In the first chapters of this course, our general objective has always been to get an optimal solution to some combinatorial optimization problem. While we often had to accept a relaxation or a bound as a starting point, we then developed methods to refine the relaxation in order to get closer to the optimum, in the hope that we would be able to reach the optimum at some point.

However, in real life applications, we will regularly have to deal with very hard and very large problem instances which may require a solution in some very limited amount of time. Consider, e. g., the sequencing of incoming approaches at an airport. Due to possible delays, weather and other external influences, the airplanes will usually not arrive in exactly the order and at exactly the time that they were scheduled for. Fortunately, fairly good predictions for arrival times are available once a flight has actually departed, thus we possess reliable information a few hours in advance. This is often enough to employ an exact algorithm that (provably) yields an optimal solution. On the other hand, predictions might be subject to change due to local weather conditions, requiring an optimization algorithm that delivers a result within minutes. In such a case, one might not be able to compute an optimal solution, thus algorithms that deliver a "good" solution fast are needed.

Another example is the classical *burglar problem*: A burglar breaks into a house and takes a look at the valuables he finds there, quickly assessing both the value and the weight of all items. To get the most valuable booty he can carry, he will have to solve a knapsack problem. However, a police siren can be heard in the distance, so the burglar only has very little time and cannot afford to sit down and solve an $\mathcal{NP}$-complete problem. What he needs is an algorithm that guarantees him at least a good value of his booty within a very short time span for calculations. (This problem is also known as the *lazy student's problem*: A student decides to start studying for the exam the night before and needs to review items that are likely to be in the exam within the available time. As she cannot afford to loose time on planning, she needs an algorithm that will give her a good turnout without requiring time-consuming calculations.)

3.1 Constant Factor Approximation

There are two classes of algorithms designed to yield good solutions in a short time that are often confused a little, approximation algorithms and heuristics. In this course, the distinction will be this: An approximation algorithm is a *polynomial time* algorithm that yields a solution within some provable *approximation guarantee*, i. e. the algorithm always delivers a solution whose deviation from the optimal solution is bounded by some factor that is known *à priori*.

Definition 3.1 (k-factor approximation)
Let Π be an optimization problem (minimization or maximization) with nonnegative objective function f and let $\mathcal{A}$ be a polynomial-time algorithm that, given an instance I of Π, computes a feasible solution $\mathcal{A}(I)$ to the problem. Then $\mathcal{A}$ is called a *k-approximation algorithm* for the problem Π for some $k \geq 1$, if

$$\frac{1}{k} \cdot f(\mathrm{OPT}(I)) \leq f(\mathcal{A}(I)) \leq k \cdot f(\mathrm{OPT}(I))$$

for all instances I of Π, where $\mathrm{OPT}(I)$ denotes some optimal solution for the instance I. We also say that $\mathcal{A}$ has *performance ratio/guarantee k*.

In some cases, the value of k may also depend on problem parameters, but we would usually not call such an algorithm a constant factor approximation. Note that the first inequality in the above definition applies to maximization problems, while the second applies to minimization problems.

3.2 Approximating Knapsack

We start with an example of an approximation algorithm for the knapsack problem. Recall the notation for a knapsack problem on n items used in the previous chapter:

$$\begin{aligned} \max\ & c^T x \\ & a^T x \leq \beta \\ & x \in \{0,1\}^n \end{aligned}$$

Here, $c \in \mathbb{Q}+^n$ is the values vector, $a \in \mathbb{N}^n$ are the weights and $\beta \in \mathbb{N}$ denotes the maximum total weight. We will assume throughout that $a_i \leq \beta$ for all i (otherwise, the corresponding items violating this inequality may be dropped beforehand as they can never be part of a feasible solution) and $a([n]) > \beta$ (otherwise $[n]$ is the optimal solution).

We are going to investigate a very simple, greedy-like approach to solving the knapsack problem: Order the items by relative value, i. e. by c_i/a_i, and then take them into the solution greedily,

starting with highest relative value, until the first item is rejected, see 3.1. We will first show that this simple approach does not quite work. A closer look at our counterexample will then yield a slightly modified algorithm that does grant a constant factor approximation.

Algorithm 3.1: A simple greedy algorithm for the knapsack problem.

Input: $n \in \mathbb{N}$, values $c \in \mathbb{N}^n$, weights $a \in \mathbb{N}^n$, weight limit $\beta \in \mathbb{N}$
Output: a feasible solution $x \in \{0,1\}^n$ of the knapsack problem

```
Sort items such that c_1/a_1 ≥ c_2/a_2 ≥ ··· ≥ c_n/a_n
Set x ← 0
Set i ← 1
while a^T x + a_i < β do
    Set x_i ← 1
    i ← i + 1
end
return x
```

To be able to analyze this algorithm in a more concise form, let us first express it in a different way. After sorting, we basically search for the first index $i' \in [n]$ where the total weigth up to that point exceeds the weight limit β. The solution is then the subset $\{1, \dots, i'-1\}$:

Proposition 3.2
Let (a, c, β) be an instance of the knapsack problem on $n \geq 2$ items such that

$$\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \cdots \geq \frac{c_n}{a_n}$$

$$\text{and set} \quad i' := \min\left\{ i \in [n] : \sum_{j=1}^{i} a_i > \beta \right\}.$$

Then choosing the feasible solution $\{1, \dots, i'-1\}$ does not constitute a k-approximation algorithm for the knapsack problem for any $k \geq 1$.

✎ Proof

Proof.
The algorithm outlined above is obviously polynomial (sorting can be done in $O(n \ln n)$ and finding i' requires going through the items once, so it is $O(n)$). Suppose there was some constant k such that the above algorithm yielded a k-factor approximation. Consider an instance I with two items where $\beta = k+1$, $a_1 = 1$, $c_1 = 1+\varepsilon$, $a_2 = k+1$, $c_2 = k+1$. Then the sorting is already correct and the algorithm would pick the solution $\{1\}$ with an overall value of $1+\varepsilon$ for some $\varepsilon > 0$, while $\{2\}$ is a much better solution

having a total value of $k + 1$. Let $\mathcal{A}$ denote the above algorithm, then by assumption, our algorithm should satisfy

$$\frac{1}{k} f(\mathrm{OPT}(I)) \leq f(\mathcal{A}(I)) \Rightarrow \frac{1}{k}(k+1) \leq 1 + \varepsilon$$

which is contradiction for sufficiently small values of $\varepsilon > 0$. □

The problem here is that our simple algorithm acts a little short-sighted: It greedily collects all the "relatively good" items without considering the possibility that saving some space might be beneficial later on. Our counterexample could simply be dealt with if we supplied the algorithm with just a little look-ahead capabilities: Instead of just taking $\{1, \dots, i' - 1\}$ as the solution, we should have had a look at $\{i'\}$ and taken that if it was the better choice. As it turns out, this simple change is enough to get a constant-factor approximation algorithm.

Algorithm 3.2: A 2-approximation algorithm for the knapsack problem.

Input: $n \in \mathbb{N}$, values $c \in \mathbb{N}^n$, weights $a \in \mathbb{N}^n$, weight limit $\beta \in \mathbb{N}$
Output: a feasible solution $x \in \{0, 1\}^n$ of the knapsack problem

```
Sort items such that c_1/a_1 ≥ c_2/a_2 ≥ ··· ≥ c_n/a_n
Set x ← 0
Set i ← 1
while a^T x + a_i < β do
    Set x_i ← 1
    i ← i + 1
end
if c^T x > c_i then
    return x
else
    return u_i
end
```

Proposition 3.3
Let (a, c, β) be an instance of the knapsack problem on $n \geq 2$ items such that

$$\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \cdots \geq \frac{c_n}{a_n}$$

$$\text{and set} \quad i' := \min \left\{ i \in [n] : \sum_{j=1}^{i} a_i > \beta \right\}.$$

Then choosing either $\{1, \dots, i' - 1\}$ or $\{i'\}$ as the solution, depending on which of the two has the higher total value, constitutes a 2-approximation algorithm for the knapsack problem.

✎ Proof

see [**KorteVygen:2012**]

Proof.
The algorithm is obviously polynomial again, the only change being one additional

comparison at the end of the algorithm. Let $I = (a, c, \beta)$ denote an instance of the knapsack problem. We claim that the following x^* denotes an optimal solution of the fractional knapsack problem (i. e., we are allowed to take fractional parts of all the items):

$$\begin{aligned} x_i^* &:= 1 && \text{for } i = 1, \ldots, i' - 1 \\ x_{i'}^* &:= \frac{1}{a_{i'}}(\beta - \sum_{j=1}^{i'-1} a_j) \\ x_i^* &:= 0 && \text{for } i = 1i' + 1, \ldots, n \end{aligned}$$

To see this, assume there was a better solution. This would have to include higher values for x_i^* for some $i \in \{i', i' + 1, \ldots, n\}$ and hence lower values for some x_i^* with $i \in \{1, \ldots, i' - 1\}$. As the value of the solution is

$$c(x^*) = \sum_{i=1}^{n} \frac{c_i}{a_i}(x_i^* a_i)$$

this would have to result in a solution with lower total value, a contradiction.

Thus, an upper bound on the optimal objective value for the knapsack problem is given by $C := \sum_{i=1}^{i'} c_i$, and the better of the two solutions $\{1, \ldots, i' - 1\}$ and $\{i'\}$ achieves at least half of that value C, making our algorithm $\mathcal{A}$ a 2-approximation algorithm:

$$f(\mathcal{A}(I)) \geq \frac{1}{2} f(\mathrm{OPT}(I)) \qquad \square$$

3.3 Approximating the Traveling Salesman Problem

In this section, we will consider the traveling salesman problem specified by an undirected graph $G = (V, E)$ and an edge weight function $c : E \to \mathbb{N}$. We will generally assume that the graph G has n vertices and is complete, as we can always add missing edges and give them a weight of "∞" (or a prohibitively high weight that would enable us to recognize when such an additional edge has been used in a solution). This way, we will have to worry about the existence of an optimal solution, which simplifies our setting a little. Unfortunately, this chapter starts off with some bad news regarding approximation algorithms for the traveling salesman problem.

Theorem 3.4 (Sahni and Gonzales, 1976)
There is no k-factor approximation algorithm for the traveling salesman problem for any $k \geq 1$, unless $\mathcal{P} = \mathcal{NP}$.

The proof is based on the established hardness of the Hamilton Circuit Problem, thus we will briefly review this problem.

Problem 3.6 (Hamilton Circuit)
Input: a graph $G = (V, E)$

Question: Does G contain a Hamilton circuit?

This problem is known to be $\mathcal{NP}$-hard (for a proof see [**Karp:1972**]). We will basically show that if we could approximate TSP by some constant factor, then this could be used to solve Hamilton Circuit. The main idea of the proof is to set edge weights on a complete graph in such a way that the objective value will then tell us whether the original graph did have a Hamilton circuit or not. The proof will be discussed in the tutorials. For those of you who do not attend, it can also be found in [**SahniGonzalez:1976**] or [**KorteVygen:2012**].

So, at least for the general case, there is no hope of an approximation algorithm for the traveling salesman problem. However, for more restricted cases, we might still be lucky. In practical applications, the edge weight function often represents distances and is thus defined by some metric. Such weights yield the Metric TSP as a special case.

Problem 3.7 (Metric TSP)
Input: a complete graph $K_n = ([n], E)$ on n nodes with an edge weight function $c : E \to \mathbb{Q}_+$ that satisfies the *triangle inequalities*, i. e.

$$c(x, y) \leq c(x, z) + c(z, y) \quad \text{for all } x, y, z \in [n].$$

Task: Find a Hamilton circuit of minimum total weight in K_n.

The metric version of TSP is still $\mathcal{NP}$-hard, a proof can be found in [**Karp:1972**] (in fact, the original proof for $\mathcal{NP}$-hardness of TSP still works in the case of Metric TSP). In contrast to the general case, however, we can now construct constant-factor approximations. We will get to know two approximation algorithms for Metric TSP which share the same basic idea:

- Find a minimum spanning tree T in the graph G.
- Add some edges to T to get a Eulerian graph T'.
- Find a Eulerian walk in the graph T' and use that to construct a traveling salesman tour by visiting the nodes in the order they are visited in the Eulerian walk. Take shortcuts if a node would be visited a second time.

✎ Metric TSP

Can you already see where the "metric" property of the edge weights will be important in this algorithm? Taking shortcuts will only work if they really "cut short" – this is precisely where the triangle inequality is needed.

The difference between the two algorithms is mainly the way edges are added to make the graph T Eulerian. Let us start with the simpler algorithm that simply doubles all the edges – that way, the degree of every node becomes even, thus the graph is Eulerian.

Algorithm 3.3: The double-tree algorithm for Metric TSP.

Input: an instance $K_n = ([n], E)$ with edge weights $c : E \to \mathbb{Q}_+$ of Metric TSP
Output: a TSP tour

1 Determine a minimum weight spanning tree $T \subset E$ of K_n.
2 Define the graph $G' = ([n], E')$ by setting $E' := \{(e, 1) : e \in T\} \cup \{(e, 2) : e \in T\}$, i. e. G' contains two identical copies of each edge in T
3 Find a Eulerian walk in G'. Let $(v_1, \ldots, v_n)$ the nodes of G in the order they appear in the Eulerian walk, starting with $v_1 = 1$.
4 **return** *tour* $(v_1, \ldots, v_n)$

Theorem 3.9
Algorithm 3.3 is a 2-approximation algorithm for Metric TSP.

✎ Proof

see [**Papadimitriou:1982te**; **KorteVygen:2012**]

Proof.
Let $K_n = ([n], E)$ and a weight function $c : E \to \mathbb{Q}_+$ constitute an instance of Metric TSP and let T be a minimum weight spanning tree in K_n with respect to c as computed in the algorithm. Further, let OPT denote an optimal solution of the TSP and let τ be the tour returned by the algorithm. As deleting any edge from OPT yields a spanning tree of K_n and as all edge weights are nonnegative, we have

$$c(T) \leq c(\mathrm{OPT}).$$

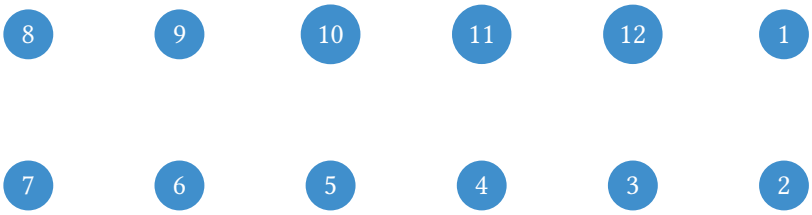
The Eulerian walk computed in the algorithm traverses every edge in T twice, so its weight is exactly equal to $2c(T)$. The tour τ then uses either the same edges or shortcuts some of them, which by the triangle inequality means

$$c(\tau) \leq 2c(T) \leq 2c(\mathrm{OPT}.$$

This establishes the proposed approximation ration of 2. Furthermore, both minimum spanning tree and Eulerian walks can be computed in polynomial time, and the computation of the tour τ then requires an additional time of $O(n)$. In total, the running time of the algorithm is polynomial, thus proving the theorem. □

A natural question to ask is whether the bound of 2 that we have obtained is in fact tight, i. e. whether there is an instance of Traveling Salesman where the algorithm yields a solution whose objective value is really twice that of an optimal solution. The following example shows that this is indeed the case.

The Double Tree Bound is Tight



Example 3.10
Consider a $n \times 2$-grid in the Euclidean plane $\mathbb{R}^2$ with edge length 1 and Euclidean lengths on all edges, see figure 3.1a. An optimal tour (as shown in the figure) has length $2n$ (a shorter tour that visits all nodes is not possible).

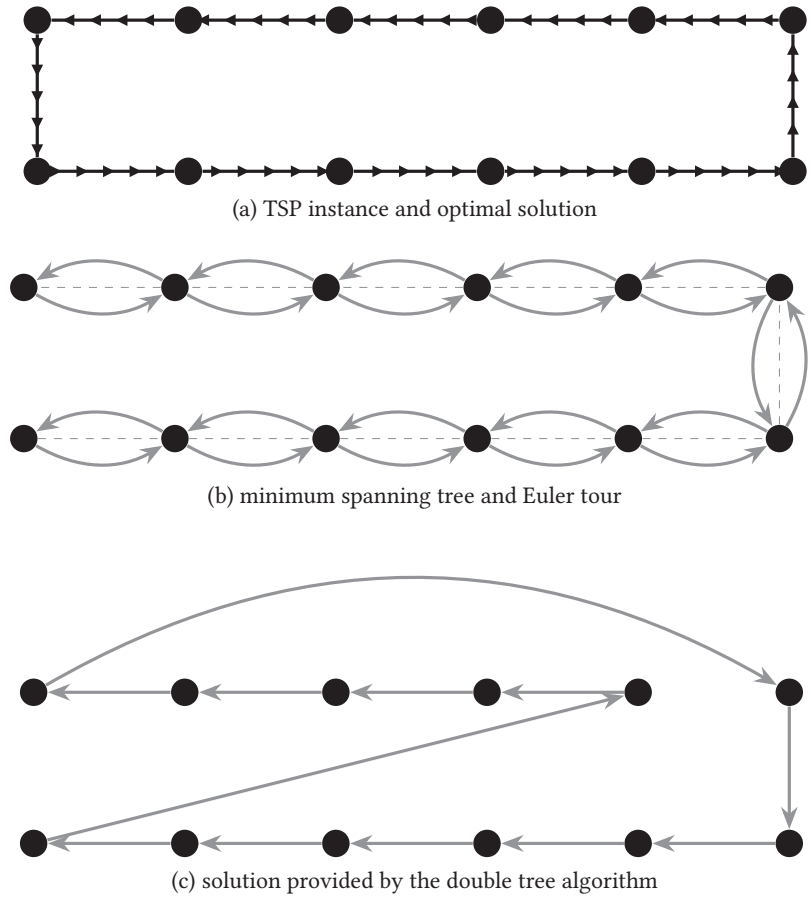
On the other hand, consider the spanning tree and corresponding Euler tour depicted in figure 3.1b. Starting at the upper right vertex and following the Euler tour downwards results in the MST solution depicted in figure 3.1c. For a $2 \times n$-grid with euclidean distances, this yields a TSP tour of length

$$3n - 3 + \sqrt{1 + (n-2)^2} > 4n - 5.$$

Thus, the approximation ratio of this solution is

$$\frac{4n-5}{2n} = 2 - \frac{5}{2n} \to 2 \quad \text{for } n \to \infty. \qquad \diamond$$

While the double three algorithm is pretty easy to implement and does yield decent results, there is one step that could easily be replaced by some better approach: The doubling of the tree is a somewhat crude way to make the graph Eulerian. Usually, most of the nodes in a tree will already have even degree, so it is only necessary to add edges to those which have odd degree with respect to T. Due to the "handshake lemma" (that we know from other courses on graph theory, see [**Taraz:2012**; **MatousekNesetril:2008**] for reference), the number of odd degree



(a) TSP instance and optimal solution

(b) minimum spanning tree and Euler tour

(c) solution provided by the double tree algorithm

Figure 3.1: A TSP instance where the double tree algorithm yields a solution realizing approximation ratio 2.

nodes in a graph is always even, so a matching of just these nodes suffices to get a Eulerian

graph. This is the main idea of *Christofides' Algorithm* (see [**Christofides:1976**]).

Algorithm 3.4: Christofides' algorithm for Metric TSP.

Input: an instance $K_n = ([n], E)$ with edge weights $c : E \to \mathbb{Q}_+$ of Metric TSP
Output: a TSP tour

1 Determine a minimum weight spanning tree $T \subset E$ of K_n.
2 Let $V' \subset V$ be the set of nodes having odd degree with respect to T.
3 Compute a minimum weight perfect matching $M \subset E$ in V'.
4 Define the graph $G' = ([n], E')$ by setting $E' := T \cup M$, i. e. G' contains the tree edges and the matching edges.
5 Find a Eulerian walk in G'. Let $(v_1, \ldots, v_n)$ the nodes of G in the order they appear in the Eulerian walk, starting with $v_1 = 1$.
6 **return** *tour* $(v_1, \ldots, v_n)$

Theorem 3.11
Algorithm 3.4 is a $\frac{3}{2}$-approximation algorithm for Metric TSP.

✎ Proof

see [**Christofides:1976**; **Papadimitriou:1982te**; **KorteVygen:2012**]

Proof.
Let $K_n = ([n], E)$ and a weight function $c : E \to \mathbb{Q}_+$ constitute an instance of Metric TSP. Further, let T be a minimum weight spanning tree in K_n with respect to c, $V' \subset V$ the subset of odd-degree nodes in G with respect to T and $M \subset E$ a minimum weight perfect matching of the nodes in V' with respect to c as computed in the algorithm. As always, we will denote an optimal solution of the TSP by OPT and τ will be the tour returned by the algorithm. As deleting any edge from OPT yields a spanning tree of K_n and as all edge weights are nonnegative, we have

$$c(T) \leq c(\text{OPT}) \quad \text{as before.}$$

Let $v'_1, \ldots, v'_k \in V'$ be the nodes in V' in the order in which they are traversed in the optimal solution OPT and consider the two matchings

$$M_1 = \left\{(v'_1, v'_2), (v'_3, v'_4). \ldots, (v'_{k-1}, v'_k)\right\}$$
$$\text{and} \quad M_2 = \left\{(v'_2, v'_3), (v'_4, v'_5). \ldots, (v'_k, v'_1)\right\}.$$

Together, these matchings constitute a circuit that traverses V' by shortcutting the optimal tour OPT, thus by the triangle inequality

$$c(M_1) + c(M_2) \leq c(\text{OPT}),$$

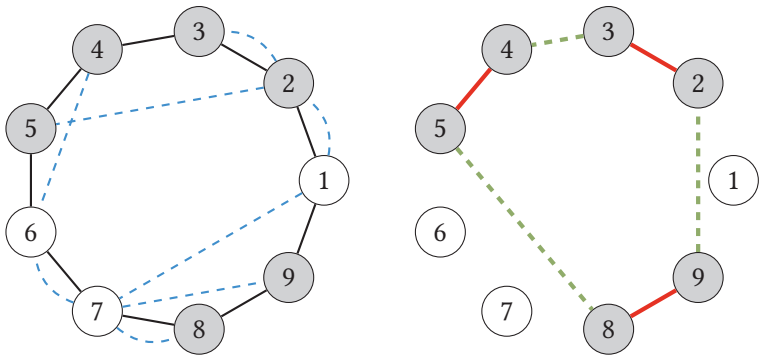
and as M is a minimum weight perfect matching of V', its weight is less or equal to both $c(M_1)$ and $c(M_2)$, consequently

$$2c(M) \leq c(M_1) + c(M_2) \leq c(\mathrm{OPT}) \quad \Rightarrow \quad c(M) \leq \frac{1}{2}c(\mathrm{OPT}).$$

Together with the above inequality for the spanning tree T, we get (again using the triangle inequality)

$$c(\tau) \leq c(T) + c(M) \leq c(\mathrm{OPT}) + \frac{1}{2}c(\mathrm{OPT}) \leq \frac{3}{2}c(\mathrm{OPT}).$$

Minimum spanning tree, matching and Eulerian walk can again be computed in polynomial time, and the same is true for the tour τ itself, completing the proof. □



(a) optimal tour (black) and minimum spanning tree (dashed blue)

(b) matchings M_1 (thick red) and M_2 (dashed, thick green)

Figure 3.2: Illustration for the proof of the approximation ratio for Christofides' algorithm. Odd-degree nodes are marked in gray color.

It can be shown that the approximation ratios that we proved for the two algorithms are tight, i. e. there are instances where the solution computed by the algorithms is in fact not better than the approximation ratio guarantees. We will have a look at this question in the tutorials. Also, there are a number of approximation algorithms for TSP that we will not cover in this course, e. g. the cheapest insert algorithm, which is a constructive algorithm (it starts with an empty tour and repeatedly adds the node that can be inserted into the tour at minimum cost – this yields a 2-approximation).

3.4 Approximation Schemes

Instead of dwelling on examples of constant factor approximation, we will now take our exploration one step further. While constant factor approximation does give us a nice guarantee on the worst case performance of our algorithmic approach, they only work up to the constant factor that we can prove. Even if we would be willing to put some more work into the solution process, there is no easy (i. e., polynomial time) way to improve upon the solution. What we would ideally like to have is a class of approximation algorithms where we can choose an approximation factor (> 1, of course) and then get an algorithm that adheres to that factor, so that we may balance the amount of work againt the required precision for ourselves. This is exctly what *approximation schemes* are for.

Definition 3.12 (PTAS, FPTAS)
Let Π be an optimization problem (minimization or maximization) with nonnegative objective function f and let $\mathcal{A}$ be an algorithm that accepts as input an instance I of Π and a number $\varepsilon > 0$ and returns a $(1 + \varepsilon)$-approximate solution to the instance I.