4. Approximation Algorithms

4.1 Introduction

Currently, approximation algorithms seem to be the most successful approach for solving hard optimization problems. Immediately after introducing NP-hardness (completeness) as a concept for proving the intractability of computing problems, the following question was posed:

> *"If an optimization problem does not admit any efficient*[1] *algorithm computing an optimal solution, is there a possibility to efficiently compute at least an approximation of the optimal solution?"*

Already in the middle of the 1970s a positive answer was given for several optimization problems. It is a fascinating effect if one can jump from exponential complexity (a huge inevitable amount of physical work for inputs of a realistic size) to polynomial complexity (a tractable amount of physical work) due to a small change in the requirements – instead of an exact optimal solution one demands a solution whose cost differs from the cost of an optimal solution by at most $\varepsilon\%$ of the cost of an optimal solution for some $\varepsilon > 0$. This effect is very strong, especially, if one considers problems for which these approximation algorithms are of large practical importance. It should not be surprising when one currently considers an optimization problem to be tractable if there exists a polynomial-time approximation algorithm that solves it with a reasonable relative error.

The goals of this chapter are the following:

(i) To give the fundamentals of the concept of the approximation of optimization problems, including the classification of the optimization problems according to their polynomial-time approximability.

(ii) To present some transparent examples of approximation algorithms in order to show successful techniques for the design of efficient approximation algorithms.

(iii) To give some of the basic ideas of methods for proving lower bounds on polynomial-time inapproximability.

[1] Polynomial time

To reach these goals we organize this chapter as follows. Section 4.2 provides the definition of fundamental notions like δ-approximation algorithms, polynomial-time approximation scheme, dual approximation algorithms, stability of approximation, and a fundamental classification of hard optimization problems according to their polynomial-time approximability (inapproximability).

Section 4.3 includes several examples of polynomial-time approximation algorithms. We are very far from trying to present a survey on approximation algorithms for fundamental optimization problems. Rather, we prefer to give algorithms that transparently present some fundamental ideas for the design of approximation algorithms. We shall see that the classical algorithm design methods like greedy algorithms, local algorithms, and dynamic programming can be successful in the design of approximation algorithms, too. On the other hand, we also present new methods and concepts that are specific for the design of efficient approximation algorithms. The main ones are the method of dual approximation and the concept of the stability of approximation.

Section 4.4 is devoted to readers who are also interested in the theory and may be skipped by anybody who is interested in the algorithm design only. Here, an introduction to the concepts for proving lower bounds on polynomial-time inapproximability of optimization problems (under the assumption P $\neq$ NP or a similar one) is given. The approaches considered here contribute to the classification of optimization problems according to the hardness of getting an approximate solution. This topic is of practical importance, too, because the information provided could be very helpful for the choice of a suitable algorithmic approach to solve the given problem in a concrete application.

4.2 Fundamentals

4.2.1 Concept of Approximation Algorithms

We start with the fundamental definition of approximation algorithms. Informally and roughly, an approximation algorithm for an optimization problem is an algorithm that provides a feasible solution whose quality does not differ too much from the quality of an optimal solution.

Definition 4.2.1.1 *Let $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ be an optimization problem, and let A be a consistent algorithm for U. For every $x \in L_I$, the* **relative error $\varepsilon_A(x)$ of A on x** *is defined as*

$$\varepsilon_A(x) = \frac{|cost(A(x)) - Opt_U(x)|}{Opt_U(x)}.$$

For any $n \in \mathbb{N}$, we define **the relative error of A** *as*

$$\varepsilon_A(n) = \max\{\varepsilon_A(x) \mid x \in L_I \cap (\Sigma_I)^n\}.$$

For every $x \in L_I$, the **approximation ratio $R_A(x)$ of A on x** *is defined as*

$$R_A(x) = \max\left\{\frac{cost(A(x))}{Opt_U(x)}, \frac{Opt_U(x)}{cost(A(x))}\right\} = 1 + \varepsilon_A(x).$$

For any $n \in \mathbb{N}$, we define the **approximation ratio of A** *as*

$$R_A(n) = \max\{R_A(x) \mid x \in L_I \cap (\Sigma_I)^n\}.$$

For any positive real $\delta > 1$, we say that A is a **δ-approximation algorithm for U** *if $R_A(x) \leq \delta$ for every $x \in L_I$.*

For every function $f : \mathbb{N} \to \mathbb{R}^+$, we say that A is an **$f(n)$-approximation algorithm for U** *if $R_A(n) \leq f(n)$ for every $n \in \mathbb{N}$.*

Note that unfortunately there are many different terms used to refer to R_A in the literature. The most frequent ones, besides the term approximation ratio used here, are worst case performance, approximation factor, performance bound, performance ratio, and error ratio.

To illustrate Definition 4.2.1.1 we give an example of a 2-approximation algorithm.

Example 4.2.1.2 Consider the problem of makespan scheduling (MS). Remember that the input consists of n jobs with designated processing times $p_1, p_2, \ldots, p_n$ and a positive integer $m \geq 2$. The problem is to schedule these n jobs on m identical machines. A schedule of jobs is an assignment of the jobs to the machines. The objective is to minimize the time after which all jobs are executed, i.e., to minimize the maximum working times of the m machines.

For instance, for seven jobs of processing times $3, 2, 4, 1, 3, 3, 6$, and 4 machines, an optimal scheduling is depicted in Figure 4.1. The cost of this solution is 6.

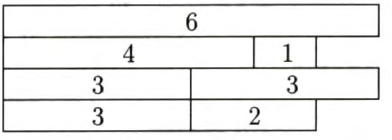

Fig. 4.1.

Now we present a simple greedy algorithm for MS and show that it is a 2-approximation algorithm. The idea is first to sort the values $p_1, p_2, \ldots, p_n$ in a nonincreasing order and then to assign the largest nonrealized job to one of the first machines that would be free.

Algorithm 4.2.1.3 GMS (GREEDY MAKESPAN SCHEDULE)

Input: $I = (p_1, \ldots, p_n, m)$, n, m, $p_1, \ldots, p_n$ positive integers and $m \geq 2$.

Step 1: Sort $p_1, \ldots, p_n$.
To simplify the notation we assume $p_1 \geq p_2 \geq \cdots \geq p_n$ in the rest of the algorithm.

Step 2: **for** $i = 1$ **to** m **do**
begin $T_i := \{i\}$;
$Time(T_i) := p_i$
end
{In the initialization step the m largest jobs are distributed to the m machines. At the end, T_i should contain the indices of all jobs assigned to the ith machine for $i = 1, \ldots, m$.}

Step 3: **for** $i = m + 1$ **to** n **do**
begin compute an l such that
$Time(T_l) := \min\{Time(T_j) | 1 \leq j \leq m\}$
$T_l := T_l \cup \{i\}$;
$Time(T_l) := Time(T_l) + p_i$
end

Output: $(T_1, T_2, \ldots, T_m)$.

Now, let us prove that GMS is a 2-approximation algorithm for MS. Let $I = (p_1, p_2, \ldots, p_n, m)$ with $p_1 \geq p_2 \geq \cdots \geq p_n$ as input. First, we observe the straightforward fact that

$$Opt_{\text{MS}}(I) \geq p_1 \geq p_2 \geq \cdots \geq p_n. \tag{4.1}$$

Clearly,

$$Opt_{\text{MS}}(I) \geq \frac{\sum_{i=1}^{n} p_i}{m} \tag{4.2}$$

because $\left(\sum_{i=1}^{n} p_i\right)/m$ is the average work time over all m machines. Since p_k is the smallest value among $p_1, p_2, \ldots, p_k$ and $\frac{1}{k}\sum_{i=1}^{k} p_i$ is the average value over $p_1, p_2, \ldots, p_k$,

$$p_k \leq \frac{\sum_{i=1}^{k} p_i}{k} \tag{4.3}$$

for every $k = 1, 2, \ldots, n$. Now we distinguish two possibilities according to the relation between n and m.

(1) Let $n \leq m$.
Since $Opt_{\text{MS}}(I) \geq p_1$ (4.1) and $cost(\{1\}, \{2\}, \ldots, \{n\}, \emptyset, \ldots, \emptyset) = p_1$, GMS has found an optimal solution and so the approximation ratio is 1.

(2) Let $n > m$.

Let T_l be such that $cost(T_l) = \sum_{r \in T_l} p_r = cost(\mathrm{GMS}(I))$, and let k be the largest index in T_l. If $k \leq m$, then $|T_l| = 1$ and so $Opt_{\mathrm{MS}}(I) = p_1 = p_k$ and $\mathrm{GMS}(I)$ is an optimal solution.

Now, assume $m < k$. Following Figure 4.2 we see that

$$Opt_{\mathrm{MS}}(I) \geq cost(\mathrm{GMS}(I)) - p_k \tag{4.4}$$

because of $\sum_{i=1}^{k-1} p_i \geq m \cdot [cost(\mathrm{GMS}(I)) - p_k]$ and (4.2).

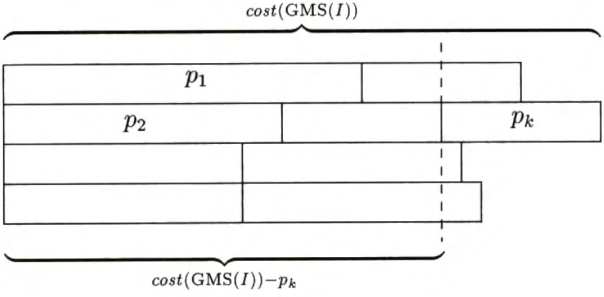

Fig. 4.2.

Combining the inequalities (4.3) and (4.4) we obtain

$$cost(\mathrm{GMS}(I)) - Opt_{\mathrm{MS}}(I) \underset{(4.4)}{\leq} p_k \underset{(4.3)}{\leq} \left(\sum_{i=1}^{k} p_i \right) \Big/ k. \tag{4.5}$$

Finally, we bound the relative error by the following calculation:

$$\frac{cost(\mathrm{GMS}(I)) - Opt_{\mathrm{MS}}(I)}{Opt_{\mathrm{MS}}(I)} \underset{\substack{(4.5)\\(4.2)}}{\leq} \frac{\left(\sum_{i=1}^{k} p_i\right)/k}{\left(\sum_{i=1}^{n} p_i\right)/m} \leq \frac{m}{k} < 1.$$

□

Exercise 4.2.1.4 Change the greedy strategy of the algorithm GMS to an arbitrary choice, i.e., without sorting $p_1, p_2, \ldots, p_n$ (removing Step 1 of GMS), assign the jobs in the on-line manner[2] as described in Step 2 and 3 of GMS. Prove that this simple algorithm, called GRAHAM'S ALGORITHM, is a 2-approximation algorithm for MS, too. □

Exercise 4.2.1.5 Find, for every integer $m \geq 2$, an input instance I_m of MS such that $R_{\mathrm{GMS}}(I) = \frac{cost(\mathrm{GMS}(I))}{Opt_{\mathrm{MS}}(I)}$ is as large as possible. □

[2]Whenever a machine becomes available (free), the next job on the list is assigned to begin processing on that machine.

In Section 4.3 we present several further approximation algorithms. The simplest examples illustrating Definition 4.2.1.1 are the 2-approximation algorithms for the vertex problem (Algorithm 4.3.2.1) and for the simple knapsack problem (Algorithm 4.3.4.1).

Usually one can be satisfied if one can find a δ-approximation algorithm for a given optimization problem with a conveniently small δ. But for some optimization problems we can do even better. For every input instance x, the user may choose an arbitrarily small relative error ε, and we can provide a feasible solution to x with a relative error at most ε. In such a case we speak about approximation schemes.

Definition 4.2.1.6 *Let $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ be an optimization problem. An algorithm A is called a* **polynomial-time approximation scheme (PTAS) for U**, *if, for every input pair $(x, \varepsilon) \in L_I \times \mathbb{R}^+$, A computes a feasible solution $A(x)$ with a relative error at most ε, and $Time_A(x, \varepsilon^{-1})$ can be bounded by a function[3] that is polynomial in $|x|$. If $Time_A(x, \varepsilon^{-1})$ can be bounded by a function that is polynomial in both $|x|$ and ε^{-1}, then we say that A is a* **fully polynomial-time approximation scheme (FPTAS) for U**.

A simple example of a PTAS is Algorithm 4.3.4.2 for the simple knapsack problem, and Algorithm 4.3.4.9 is a FPTAS for the knapsack problem.

In the typical case the function $Time_A(x, \varepsilon^{-1})$ grows in both $|x|$ and ε^{-1}. This means that one has to pay for the decrease in the relative error (increase of the quality of the output) by an increase in the time complexity (amount of computer work). The advantage of PTASs is that the user has the choice of ε in this tradeoff of the quality of the output and of the amount of computer work $Time_A(x, \varepsilon^{-1})$. FPTASs are very convenient[4] because $Time_A(x, \varepsilon^{-1})$ does not grow too quickly with ε^{-1}.

4.2.2 Classification of Optimization Problems

Following the notion of approximability we divide the class NPO of optimization problems into the following five subclasses:

NPO(I): Contains every optimization problem from NPO for which there exists a FPTAS.

{In Section 4.3 we show that the knapsack problem belongs to this class.}

NPO(II): Contains every optimization problem from NPO that has a PTAS.

{In Section 4.3.4 we show that the makespan scheduling problem belongs to this class.}

[3]Remember that $Time_A(x, \varepsilon^{-1})$ is the time complexity of the computation of the algorithm A on the input (x, ε).

[4]Probably a FPTAS is the best that one can have for a NP-hard optimization problem.

NPO(III): Contains every optimization problem $U \in$ NPO such that

(i) there is a polynomial-time δ-approximation algorithm for some $\delta > 1$, and

(ii) there is no polynomial-time d-approximation algorithm for U for some $d < \delta$ (possibly under some reasonable assumption like P $\neq$ NP), i.e., there is no PTAS for U.

{The minimum vertex cover problem, MAX-SAT, and $\triangle$-TSP are examples of members of this class.}

NPO(IV): Contains every $U \in NPO$ such that

(i) there is a polynomial-time $f(n)$-approximation algorithm for U for some $f : \mathbb{N} \to \mathbb{R}^+$, where f is bounded by a polylogarithmic function, and

(ii) under some reasonable assumption like P $\neq$ NP, there does not exist any polynomial-time δ-approximation algorithm for U for any $\delta \in \mathbb{R}^+$.

{The set cover problem belongs to this class.}

NPO(V): Contains every $U \in$ NPO such that if there exists a polynomial-time $f(n)$-approximation algorithm for U, then (under some reasonable assumption like P $\neq$ NP) $f(n)$ is not bounded by any polylogarithmic function.

{TSP and the maximum clique problem are well-known members of this class.}

Note that assuming P $\neq$ NP none of the classes NPO(I), NPO(II), NPO(III), NPO(IV) or NPO(V) is empty. Thus, we have to deal with all these five types of hardness according to the approximability of problems in NPO.

4.2.3 Stability of Approximation

If one is asking for the hypothetical border between tractability and intractability for optimization problems it is not so easy to give an unambiguous answer. FPTASs and PTASs are usually very practical but we know examples, where the time complexity of a PTAS[5] is around $n^{40 \cdot \varepsilon^{-1}}$ for the input length n and this is obviously not very practical. On the other hand, some $(\log_2 n)$-approximation algorithm may be considered to be practical for some applications. What people usually consider to be intractable are problems in NPO(V). But one has to be careful with this, too. We are working with the worst case complexity, and also the relative error and the approximation ratio of an approximation algorithm

[5] We consider the PTAS for the geometrical (Euclidean) TSP.

are defined in the worst case manner. The problem instances of concrete applications may be much easier than the hardest ones, even much easier than the average ones. It could be helpful to split the set of input instances L_I of a $U \in$ NPO(V) into some (possibly infinitely many) subclasses according to the hardness of their polynomial-time approximability, and to have an efficient algorithm deciding the membership of any input instance to one of the subclasses considered. In order to reach this goal one can use the notion of the stability of approximation.

Informally, one can explain the concept of stability with the following scenario. One has an optimization problem for two sets of input instances L_1 and L_2, $L_1 \subsetneq L_2$. For L_1 there exists a polynomial-time δ-approximation algorithm A for some $\delta > 1$, but for L_2 there is no polynomial-time γ-approximation algorithm for any $\gamma > 1$ (if NP is not equal P). One could pose the following question: "Is the use of the algorithm A really restricted to inputs from L_1?" Let us consider a distance measure d in L_2 determining the distance $d(x)$ between L_1 and any given input $x \in L_2 - L_1$. Now, one can look for how "good" the algorithm A for the inputs $x \in L_2 - L_1$ is. If, for every $k > 0$ and every x with $d(x) \leq k$, A computes a $\gamma_{k,\delta}$-approximation of an optimal solution for x ($\gamma_{k,\delta}$ is considered to be a constant depending on k and δ only), then one can say that A is "(approximation) stable" according to the distance measure d.

Obviously, the idea of this concept is similar to that of the stability of numerical algorithms. But instead of observing the size of the change of the output value according to a small change of the input value, we look for the size of the change of the approximation ratio according to a small change in the specification (some parameters, characteristics) of the set of problem instances considered. If the exchange of the approximation ratio is small for every small change in the specification of the set of problem instances, then we have a stable algorithm. If a small change in the specification of the set of problem instances causes an essential (depending on the size of the input instances) increase of the relative error, then the algorithm is unstable.

The concept of stability enables us to show positive results extending the applicability of known approximation algorithms. As we shall see later, the concept motivates us to modify an unstable algorithm A in order to get a stable algorithm B that achieves the same approximation ratio on the original set of input instances as A has, but B can also be successfully used outside of the original set of input instances. This concept is useful because there are a lot of problems for which an additional assumption on the "parameters" of the input instances leads to an essential decrease in the hardness of the problem. So, such effects are the starting points for trying to partition the whole set of input instances into a spectrum of classes according to approximability.

Definition 4.2.3.1 *Let* $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ *and* $\overline{U} = (\Sigma_I, \Sigma_O, L, L, \mathcal{M}, cost, goal)$ *be two optimization problems with* $L_I \subsetneq L$. *A* **distance function for $\overline{U}$ according to L_I** *is any function* $h_L : L \rightarrow \mathbb{R}^{\geq 0}$ *satisfying the properties*

(i) $h_L(x) = 0$ *for every* $x \in L_I$, *and*

(ii) h *is polynomial-time computable.*

Let h *be a distance function for* $\overline{U}$ *according to* L_I. *We define, for any* $r \in \mathbb{R}^+$,

$$\boldsymbol{Ball_{r,h}(L_I)} = \{w \in L \mid h(w) \leq r\}.^6$$

Let A *be a consistent algorithm for* $\overline{U}$, *and let* A *be an* ε*-approximation algorithm for* U *for some* $\varepsilon \in \mathbb{R}^{>1}$. *Let* p *be a positive real. We say that* A *is* **p-stable according to** $\boldsymbol{h}$ *if, for every real* $0 < r \leq p$, *there exists a* $\delta_{r,\varepsilon} \in \mathbb{R}^{>1}$ *such that* A *is a* $\delta_{r,\varepsilon}$*-approximation algorithm for* $U_r = (\Sigma_I, \Sigma_O, L, Ball_{r,h}(L_I), \mathcal{M}, cost, goal)$.

A *is* **stable according to h** *if* A *is* p*-stable according to* h *for every* $p \in \mathbb{R}^+$. *We say that* A *is* **unstable according to** $\boldsymbol{h}$ *if* A *is not* p*-stable for any* $p \in \mathbb{R}^+$.

For every positive integer r, *and every function* $f_r : \mathbb{N} \to \mathbb{R}^{>1}$ *we say that* A *is* **$(r, f_r(n))$-quasistable according to h** *if* A *is an* $f_r(n)$*-approximation algorithm for* $U_r = (\Sigma_I, \Sigma_O, L, Ball_{r,h}(L_I), \mathcal{M}, cost, goal)$.

Example 4.2.3.2 Consider TSP $= (\Sigma_I, \Sigma_O, L, L, \mathcal{M}, cost, minimum)$ and $\triangle$-TSP $= (\Sigma_I, \Sigma_O, L, L_\triangle, \mathcal{M}, cost, minimum)$ as defined in Section 2.3. Remember that L contains all weighted complete graphs and $L_\triangle$ contains all weighted complete graphs with a weight function c satisfying the triangle inequality. A natural way to define a distance function h for TSP according to $L_\triangle$ is to measure the "distance" of any input instance from the triangle inequality. For instance, this can be done in the following ways:

For every input instance (G, c),

$$dist(G,c) = \max\left\{0, \max\left\{\frac{c(\{u,v\})}{c(\{u,p\}) + c(\{p,v\})} - 1 \,\middle|\, u,v,p \in V(G), u \neq v, u \neq p, v \neq p\right\}\right\},$$

$$dist_k(G,c) = \max\left\{0, \max\left\{\frac{c(\{u,v\})}{\sum_{i=1}^{m} c(\{p_i, p_{i+1}\})} - 1 \,\middle|\, u,v \in V(G) \text{ and } u = p_1, p_2, \ldots, p_m = v \text{ is a simple path between } u \text{ and } v \text{ of length at most } k \text{ (i.e., } m+1 \leq k)\right\}\right\}$$

for every integer $k \geq 2$, and

$$distance(G,c) = \max\{dist_k(G,c) \mid 2 \leq k \leq |V(G)| - 1\}.$$

We observe that $Ball_{r,dist}(L_\triangle)$ contains exactly those weighted graphs for which

[6] $Ball_{r,h}(L_I)$ contains all input instances whose specification "differs" at most r from the specification of the input instances in L_I.

$$c(\{u,v\}) \leq (1+r)(c(\{u,p\}) + c(\{p,v\}))$$

for all pairwise different vertices u, v, p. This is a natural extension of the triangle inequality specification. The distance function $dist_k$ says that if $dist_k(G,c) \leq r$, then the cost of the direct connection (u,v) between any two vertices u and v is not larger than $(1+r)$ times the cost of any path between u and v of length at most k. □

Exercise 4.2.3.3 Prove that the functions *dist*, $dist_k$, and *distance* (defined in Example 4.2.3.2) are distance functions for TSP according to $L_\triangle$. □

Some examples of stable and unstable algorithms according to the distance functions *dist* and *distance* for TSP are given in Section 4.3.5.

In Definition 4.2.3.1 distance functions were defined in a very general form. It should be clear that the investigation of the stability according to a distance function h is of interest only if h reasonably "partitions" the set of problem instances. The following exercise shows a distance function that is not helpful for the study of approximability of any problem.

Exercise 4.2.3.4 Let $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ and $\overline{U} = (\Sigma_I, \Sigma_O, L, L, \mathcal{M}, cost, goal) \in$ NPO. Let h_{index} be defined as follows:

(i) $h_{index}(w) = 0$ for every $w \in L_I$, and

(ii) $h_{index}(u)$ is equal to the order of u according to the canonical order of words in Σ_I^*.

Prove that

a) h_{index} is a distance function of $\overline{U}$ according to L_I.

b) For every δ-approximation algorithm A for U, if A is consistent for $\overline{U}$, then A is stable according to h_{index}. □

Exercise 4.2.3.3 shows that it is not interesting to consider a distance function h with the property

$$|Ball_{r,h}(L_I)| - |Ball_{q,h}(L_I)| \text{ is finite}$$

for every $r > q$. The distance function h' investigated later has the following additional property (called the **property of infinite jumps**):

"If $Ball_{q,h'}(L_I) \subsetneq Ball_{r,h'}(L_I)$ for some $q < r$,
then $|Ball_{r,h'}(L_I)| - |Ball_{q,h'}(L_I)|$ is infinite."

But this additional property on h' also does not need to secure any kind of "reasonability" of h' for the study of approximation stability. The best approach

to define a distance function is to take a "natural" function according to the specification of the set L_I of input instances.

Exercise 4.2.3.5 Define two optimization problems $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ and $\overline{U} = (\Sigma_I, \Sigma_O, L, L, \mathcal{M}, cost, goal)$ from NPO with infinite $|L| - |L_I|$, and a distance function h for $\overline{U}$ according to L_I such that:

(i) h has the property of infinite jumps, and

(ii) for every δ-approximation algorithm A for U, if A is consistent for $\overline{U}$, then A is stable according to h. □

We see that the existence of a stable c-approximation algorithm for U immediately implies the existence of a $\delta_{r,c}$-approximation algorithm for U_r for any $r > 0$. Note that applying the concept of stability to PTASs one can get two different outcomes. Let us consider a PTAS A as a collection of polynomial-time $(1+\varepsilon)$-approximation algorithms A_ε for every $\varepsilon \in \mathbb{R}^+$. If A_ε is stable according to a distance measure h for every $\varepsilon > 0$, then we can obtain either

(i) a PTAS for $U_r = (\Sigma_I, \Sigma_O, L, Ball_{r,h}(L_I), \mathcal{M}, cost, goal)$ for every $r \in \mathbb{R}^+$ (this happens, for instance, if $\delta_{r,\varepsilon} = 1 + \varepsilon \cdot f(r)$, where f is an arbitrary function), or

(ii) a $\delta_{r,\varepsilon}$-approximation algorithm for U_r for every $r \in \mathbb{R}^+$, but no PTAS for U_r for any $r \in \mathbb{R}^+$ (this happens, for instance, if $\delta_{r,\varepsilon} = 1 + r + \varepsilon$).

To capture these two different situations we introduce the notion of "superstability".

Definition 4.2.3.6 *Let $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ and $\overline{U} = (\Sigma_I, \Sigma_O, L, L, \mathcal{M}, cost, goal)$ be two optimization problems with $L_I \subsetneq L$. Let h be a distance function for $\overline{U}$ according to L_I, and let $U_r = (\Sigma_I, \Sigma_O, L, Ball_{r,h}(L_I), \mathcal{M}, cost, goal)$ for every $r \in \mathbb{R}^+$. Let $A = \{A_\varepsilon\}_{\varepsilon>0}$ be a* PTAS *for U.*

If, for every $r > 0$ and every $\varepsilon > 0$, A_ε is a $\delta_{r,\varepsilon}$-approximation algorithm for U_r, we say that the PTAS *A is* **stable according to h**.

If $\delta_{r,\varepsilon} \leq f(\varepsilon) \cdot g(r)$, where

(i) f and g are some functions from $\mathbb{R}^{\geq 0}$ to $\mathbb{R}^+$, and

(ii) $\lim_{\varepsilon \to 0} f(\varepsilon) = 0$,

then we say that the PTAS *A is* **superstable according to h**.

Observation 4.2.3.7 *If A is a superstable (according to a distance function h)* PTAS *for an optimization problem $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$, then A is a* PTAS *for the optimization problem $U_r = (\Sigma_I, \Sigma_O, L, Ball_{r,h}(L_I), \mathcal{M}, cost, goal)$ for any $r \in \mathbb{R}^+$.*

An example of an superstable PTAS according to a distance function h for the knapsack problem is given in Section 4.3.5.

4.2.4 Dual Approximation Algorithms

In the previous approximation concepts we have focused on searching for a feasible solution whose cost well approximates the cost of an optimal solution. Sometimes there are reasons to consider another scenario. Let us consider an optimization problem $U = (\Sigma_I, \Sigma_O, L, L, \mathcal{M}, cost, goal)$, where the constraints determining $\mathcal{M}(x)$ for every input instance $x \in L$ are either not known precisely (i.e., one has an approximation of the constraints only) or not rigid, so that a small violation of the constraints is a natural model of allowed flexibility. An example may be the simple knapsack problem (SKP), where, for any input $I = (a_1, \ldots, a_n, b)$,

$$\mathcal{M}(I) = \left\{ (x_1, \ldots, x_n) \in \{0,1\}^n \,\middle|\, \sum_{i=1}^{n} x_i a_i \leq b \right\}.$$

Now, it may be possible that some values of $a_1, \ldots, a_n, b$ (especially b) are not precisely determined and a small violation of the constraint $\sum_{i=1}^{n} x_i a_i \leq b$ to a constraint $\sum_{i=1}^{n} x_i a_i \leq b + \varepsilon$ for small ε could be accepted. In such a case one can prefer to compute a solution S whose cost is at least as good as[7] $Opt_U(I)$ by allowing $S \notin \mathcal{M}(I)$, but where S is not "too far" from the specification of the elements in $\mathcal{M}(I)$.

To formalize this idea one needs a distance function again. But this distance function has to measure the distance of a solution S from the set of feasible solutions $\mathcal{M}(I)$ in contrast to the distance function measuring a distance between input instances in the concept of approximation stability.

Definition 4.2.4.1 *Let $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ be an optimization problem. A* **constraint distance function** *for U is any function $h : L_I \times \Sigma_O^* \to \mathbb{R}^{\geq 0}$ such that*

(i) $h(x, S) = 0$ for every $S \in \mathcal{M}(x)$,

(ii) $h(x, S) > 0$ for every $S \notin \mathcal{M}(x)$, and

(iii) h is polynomial-time computable.

For every $\varepsilon \in \mathbb{R}^+$, and every $x \in L_I$, $\mathcal{M}_\varepsilon^h(x) = \{S \in \Sigma_O^* \mid h(x, S) \leq \varepsilon\}$ is the **ε-ball of $\mathcal{M}(x)$ according to h.**

To illustrate Definition 4.2.4.1 consider the bin-packing problem (BIN-P). Remember that an input instance of BIN-P is $I = (p_1, p_2, \ldots, p_n)$, where $p_i \in [0,1]$ for $i = 1, \ldots, n$, and the objective is to pack the pieces into bins of unit size in such a way that the minimum number of bins is used. For any solution $T = T_1, T_2, \ldots, T_m$ ($T_i \subseteq \{1, \ldots, n\}$ for $i = 1, \ldots, m$), one can define the constraint distance function

[7] If *goal* = *maximum* then $cost(S) \geq Opt_U(I)$, and if *goal* = *minimum* then $cost(S) \leq Opt_U(I)$.

$$h(I,T) = \max\left\{0, \max\left\{\sum_{l\in T_i} p_l \,\middle|\, i = 1,2,\ldots,m\right\} - 1\right\}.$$

If there exists an ε such that $\sum_{l\in T_i} p_l \leq 1+\varepsilon$ for all $i = 1,\ldots,m$, then $T \in \mathcal{M}^h_\varepsilon(I)$.

Definition 4.2.4.2 *Let $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ be an optimization problem, and let h be a constraint distance function for U. An optimization algorithm A for U is called an* ***h*-dual ε-approximation algorithm for U,** *if for every $x \in L_I$,*

(i) $A(x) \in \mathcal{M}^h_\varepsilon(x)$, *and*

(ii) $cost(A(x)) \geq Opt_U(x)$ *if* $goal = maximum$, *and*
$cost(A(x)) \leq Opt_U(x)$ *if* $goal = minimum$.

Definition 4.2.4.3 *Let $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ be an optimization problem, and let h be a constraint distance function for U. An algorithm A is called* ***h*-dual polynomial-time approximation scheme (*h*-dual PTAS for U)**, *if*

(i) for every input $(x,\varepsilon) \in L_I \times \mathbb{R}^+$, $A(x,\varepsilon) \in \mathcal{M}^h_\varepsilon(x)$,

(ii) $cost(A(x,\varepsilon)) \geq Opt_U(x)$ *if* $goal = maximum$, *and*
$cost(A(x,\varepsilon)) \leq Opt_U(x)$ *if* $goal = minimum$, *and*

(iii) $Time_A(x,\varepsilon^{-1})$ *is bounded by a function that is polynomial in* $|x|$.

If $Time_A(x,\varepsilon^{-1})$ *can be bounded by a function that is polynomial in both* $|x|$ *and* ε^{-1}, *then we say that A is a* ***h*-dual fully polynomial-time approximation scheme (*h*-dual FPTAS) for U**.

In Section 4.3.6 we give not only an example of a dual approximation algorithm, but we also show how one can use dual approximation algorithms in order to design ordinary approximation algorithms.

The next parts of this chapter show the usefulness of the terminology and classification defined above for the development and design of approximation algorithms for hard optimization problems.

Keywords introduced in Section 4.2

δ-approximation algorithms, PTAS, FPTAS, stability of approximation algorithms, dual approximation algorithms

Summary of Section 4.2

A small change in the specification of an optimization problem may have an essential influence on the hardness of the problem. For instance, the jump from NP-hardness to deterministic polynomial time can be caused by the following changes in the problem specification:

(i) One requires a feasible solution within a given approximation ratio instead of an exact optimal solution, or

(ii) one requires that the input instances have a "reasonable" additional property (that may be typical for inputs from some real applications).

The notion of stability of numerical algorithms can be extended to approximation stability of approximation algorithms in order to get a fine analysis of the hardness of the optimization problem considered. This analysis partially overcomes the drawbacks of the worst case complexity approach by partitioning the set of input instances into subclasses according to the polynomial-time approximability.

Dual approximation algorithms approximate the set of feasible solutions (feasibility) rather than the optimality. They are of practical interest especially if one does not have any exact (but only estimated) constraints on the feasibility of solutions.

4.3 Algorithm Design

4.3.1 Introduction

This section is an introduction to the design of approximation algorithms. The main aim is to present and to illustrate methods and ideas that have been successful in the design of approximation algorithms. We are far from trying to give an overview of the huge area of approximation algorithms here. To achieve our goal without dealing with nontrivial mathematics and too many technical details, we have to prefer the presentation of transparent examples of approximation algorithms over the presentation of the best known ones. We want to show that the fundamental general methods for algorithm design like greedy algorithms (see Algorithm 4.3.2.11 for SCP, and Algorithm 4.3.4.1 for SKP), local search algorithms (see Algorithm 4.3.3.1 for MAX-CUT), dynamic programming (see Algorithm 4.3.4.9 for KP), backtracking (see PTASs for KP in Section 4.3.4), relaxation to linear programming (Algorithm 4.3.2.19 for WEIGHT-VCP), and their combinations and modifications can be successful in the design of approximation algorithms, too. On the other hand, we explain how to develop and apply approaches that are specific for the area of the design of approximation algorithms in combinatorial optimization.

Section 4.3 is organized as follows. Section 4.3.2 is devoted to the set cover problem and its special subproblem – the vertex cover problem. We show there

that a very simple efficient heuristic provides a 2-approximation algorithm for the vertex cover problem, and that the greedy approach leads to a $(\ln n)$-approximation algorithm for the set cover problem.[8] Further, we apply the method of relaxation by reduction to linear programming in order to get a 2-approximation algorithm for the weighted minimum vertex cover problem, too.

Section 4.3.3 presents another example of a simple approach that leads to a good approximation. A simple strategy of local improvements (local search) provides a polynomial-time 2-approximation algorithm for the maximum cut problem (MAX-CUT). This means that a suitable definition of local improvements can determine such a structure on the set of feasible solutions, that every local optimum is a good approximation of total optima.

Section 4.3.4 is devoted to the knapsack problem and PTASs. First, we show that the greedy approach leads to an efficient 2-approximation algorithm for the simple knapsack problem. Finding some compromise between the greedy approach and the total search we derive a PTAS for the simple knapsack problem. Then, applying the concept of stability of approximation we show that a natural modification of the PTAS for the simple knapsack problem results in a PTAS for the general knapsack problem. Finally, we combine dynamic programming with an approximation of input values in order to get a FPTAS for the knapsack problem.

In Section 4.3.5 we study the traveling salesperson problem that is one of the hardest approximation problems.[9] First, we present two special efficient approximation algorithms that achieves constant approximation ratios for $\triangle$-TSP, i.e., for the input instances that satisfy the triangle inequality. Then, we use the concept of stability of approximation in order to partition the set of input instances of the general TSP into an infinite spectrum of classes according to an achievable polynomial-time approximability. Section 4.3.6 presents the usefulness of the concept of dual approximation algorithms. First, a dual PTAS for the bin-packing problem is designed, and then this dual PTAS is used to develop a PTAS for the makespan scheduling problem. This design method works because the nonrigid constraints of the bin-packing problem can be viewed as the objective of the makespan scheduling problem; i.e., the bin-packing problem is in some sense "dual" to the makespan scheduling problem.

4.3.2 Cover Problems, Greedy Method, and Relaxation to Linear Programming

In this section we consider the minimum vertex cover problem (MIN-VCP), the set cover problem (SCP), and the weighted minimum vertex cover problem

[8] Observe that assuming P $\neq$ NP one can prove the nonexistence of any polynomial-time $((1-\varepsilon)\cdot \ln n)$-approximation algorithm for the set cover problem, and so the efficient greedy method is the best possible approximation approach for this problem.

[9] It belongs to NPO(V).

(WEIGHT-VCP). The first algorithm presented is a 2-approximation algorithm for MIN-VCP. The idea of this algorithm is very simple. In fact, one quickly finds a maximal matching of the given graph and considers all vertices adjacent to the edges of the matching as a vertex cover. Since the cardinality of every maximal matching is a lower bound on the cardinality of any vertex cover, the proposed algorithm has a relative error of at most 1.

Algorithm 4.3.2.1

Input: A graph $G = (V, E)$.

Step 1: $C := \emptyset$ {during the computation $C \subseteq V$, and at the end C should contain a vertex cover};

$A := \emptyset$ {during the computation $A \subseteq E$ is a matching, and at the end A is a maximal matching};

$E' := E$ {during the computation $E' \subseteq E$, E' contains exactly the edges that are not covered by the actual C, and at the end $E' = \emptyset$}.

Step 2: **while** $E' \neq \emptyset$

do begin choose an arbitrary edge $\{u, v\}$ from E';
$C := C \cup \{u, v\}$;
$A := A \cup \{\{u, v\}\}$;
$E' := E' - \{$all edges adjacent to u or $v\}$

end

Output: C.

Example 4.3.2.2 We illustrate the work of Algorithm 4.3.2.1 on the input $G = (\{a, b, c, d, e, f, g, h\}, \{\{a, b\},\{b, c\}, \{c, e\}, \{c, d\}, \{d, e\}, \{d, f\}, \{d, g\}, \{d, h\}, \{e, f\},\{h, g\}\})$ depicted in Figure 4.3a. An optimal vertex cover is $\{b, e, d, g\}$ with cardinality 4. This output cannot be obtained by any choice of edges in Algorithm 4.3.2.1.

Next we consider the following choice. First $\{b, c\}$ is moved to A, $C := \{b, c\}$ and the restgraph is depicted in Figure 4.3b. Then the edge $\{e, f\}$ is chosen (see Figure 4.3c). After that $C = \{b, c, e, f\}$ and $A = \{\{b, c\}, \{e, f\}\}$. In the last step the edge $\{d, g\}$ is chosen (see Figure 4.3d). This results in the vertex cover $\{b, c, e, f, d, g\}$. □

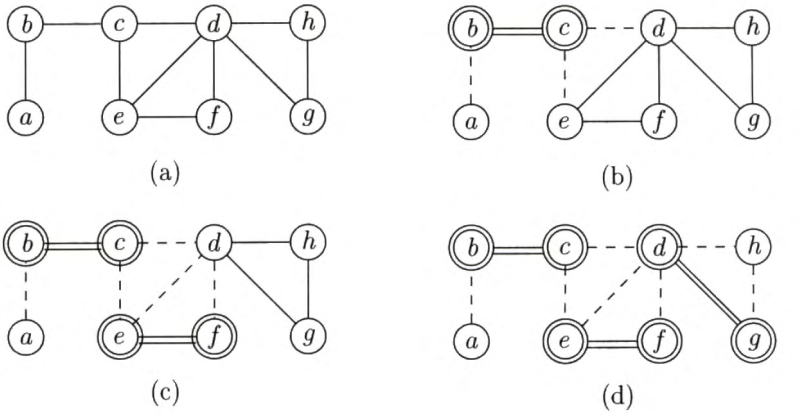

Fig. 4.3.

Exercise 4.3.2.3

(a) Find a subgraph G' of the graph G in Fig. 4.3 such that Algorithm 4.3.2.1 can output a vertex cover whose cardinality is 6 while the cardinality of an optimal vertex cover for G' is 3.

(b) Find, for every positive integer n, a Graph G_n with a vertex cover of the cardinality n, but where Algorithm 4.3.2.1 can compute a vertex cover of the size $2n$. □

Now let us analyze the complexity and the approximation ratio of Algorithm 4.3.2.1.

Lemma 4.3.2.4 *Algorithm 4.3.2.1 works in time* $O(|E|)$.

Proof. Using a convenient data structure this claim is obvious because every edge of the input graph is handled exactly once in the algorithm. □

Lemma 4.3.2.5 *Algorithm 4.3.2.1 always computes a vertex cover with an approximation ratio at most* 2.

Proof. The fact that Algorithm 4.3.2.1 outputs a vertex cover is obvious because the algorithm halts when $E' = \emptyset$ (i.e., all edges are covered by the vertices in C).

To prove that the approximation ratio is at most 2, we observe that $|C| = 2 \cdot |A|$, and that A is a matching of the input graph $G = (V, E)$. To cover the $|A|$ edges of the matching A one needs at least $|A|$ vertices. Since $A \subseteq E$, the

cardinality of every vertex cover is at least $|A|$, i.e., $Opt_{\text{MIN-VCP}}(G) \geq |A|$. Thus, $|C| = 2 \cdot |A| \leq 2 \cdot Opt_{\text{MIN-VCP}}(G)$. □

Exercise 4.3.2.6 Construct an infinite family of graphs for which Algorithm 4.3.2.1 always (independent of the choice of edges from E) computes an optimal vertex cover. □

One may say that Algorithm 4.3.2.1 is a very naive heuristic and propose a more involved choice of the vertices for the vertex cover. The most natural idea could be to use the greedy strategy that always chooses a vertex with the maximal degree in the rest graph. The claim of the following exercise shows the (maybe a little bit surprising) fact that this greedy strategy does not lead to a better approximation than the approximation ratio of Algorithm 4.3.2.1.

Exercise 4.3.2.7 (*) Consider the following greedy algorithm for VCP.

Algorithm 4.3.2.8

Input: A graph $G = (V, E)$.

Step 1: $C := \emptyset$ {at the end C should contain a vertex cover}
$E' := E$ {during the computation $E' \subseteq E$, E' contains exactly the edges that are not covered by the actual C, and at the end $E' = \emptyset$}.

Step 2: **while** $E' \neq \emptyset$
do begin choose a vertex v with the maximal degree in (V, E');
$C := C \cup \{v\}$;
$E' := E' - \{$all edges adjacent to $v\}$
end

Output: C.

Prove that the approximation ratio of Algorithm 4.3.2.8 is greater than 2. □

MIN-VCP is one of the problems for which no PTAS exists (under the assumption P $\neq$ NP). But, for every such problem, one can search for subclasses of input instances for which MIN-VCP is not that hard. One example in this direction provides the following exercise.

Exercise 4.3.2.9

(a) Design a polynomial-time algorithm for MIN-VCP when the set of input graphs is restricted to the trees.

(b) Design a polynomial-time d-approximation algorithm for MIN-VCP with $d < 2$ when the input graphs have their degree bounded by 3. □

Exercise 4.3.2.10 If $C \subseteq V$ is a vertex cover of a graph $G = (V, E)$, then $V - C$ is a clique in $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{\{u, v\} \mid u, v \in V, u \neq v\} - E$. Is it possible to use Algorithm 4.3.2.1 to get a δ-approximation algorithm for the maximum clique problem for some $\delta \in \mathbb{R}^{>1}$? □

In what follows we consider the set cover problem (SCP). This problem is in the class NPO(IV). Surprisingly, the naive greedy approach provides the best possible approximation ratio.

Algorithm 4.3.2.11

Input: $(X, \mathcal{F})$, where X is a finite set, $\mathcal{F} \subseteq 2^X$ such that $X = \bigcup_{Q \in \mathcal{F}} Q$.

Step 1: $C := \emptyset$ {during the computation $C \subseteq \mathcal{F}$ and at the end C is a set cover of $(X, \mathcal{F})$};

$U := X$ {during the computation $U \subseteq X$, $U = X - \bigcup_{Q \in C} Q$ for the actual C, and at the end $U = \emptyset$}.

Step 2: **while** $U \neq \emptyset$

do begin choose an $S \in \mathcal{F}$ such that $|S \cap U|$ is maximal;

$U := U - S$;

$C := C \cup \{S\}$

end

Output: C.

Lemma 4.3.2.12 *Algorithm 4.3.2.11 is a* $Har(\max\{|S| \mid S \in \mathcal{F}\})$*-approximation algorithm for* SCP.

Proof. Let $C = \{S_1, S_2, \ldots, S_r\}$ be the output of Algorithm 4.3.2.11, where S_i is the ith set chosen in Step 2 of Algorithm 4.3.2.11. Since $U = \emptyset$ it is obvious that C is a set cover for the input $(X, \mathcal{F})$.

For every $x \in X$, we define the **weight of x according to C** as follows:

$$\boldsymbol{Weight_C(x)} = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$

if $x \in S_i - \bigcup_{j=1}^{i-1} S_j$. Thus, $Weight_C(x)$ is determined when x is covered by Algorithm 4.3.2.11 for the first time. If one defines

$$Weight_C(C) = \sum_{x \in X} Weight_C(x)$$

then obviously $Weight_C(C) = |C| = cost(C)$. Let $C_{Opt} \in Output_{\mathrm{SCP}}(X, \mathcal{F})$ be an optimal solution with $cost(C_{Opt}) = Opt_{\mathrm{SCP}}(X, \mathcal{F})$. Our goal is to prove that the difference between $Weight_C(C)$ and $|C_{Opt}|$ is not too large relative to $|C_{Opt}| = cost(C_{Opt})$. To achieve this goal we shall prove

$$\text{for all } S \in \mathcal{F} : \sum_{x \in S} Weight_C(x) \leq Har(|S|). \tag{4.6}$$

Let, for every set $S \in \mathcal{F}$ and every $i \in \{0, 1, \ldots, |C|\}$,

$$cov_i(S) = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|.$$

Obviously, for every $S \in \mathcal{F}$, $cov_0(S) = |S|$, $cov_{|C|}(S) = 0$, and $cov_{i-1}(S) \geq cov_i(S)$ for $i = 1, \ldots, |C|$. Let k be the smallest number with the property $cov_k(S) = 0$. Since $cov_{i-1}(S) - cov_i(S)$ elements from S are covered in the ith run of Step 2,

$$\sum_{x \in S} Weight_C(x) = \sum_{i=1}^{k} (cov_{i-1}(S) - cov_i(S)) \cdot \frac{1}{\left| S_i - \bigcup_{j=1}^{i-1} S_j \right|}. \tag{4.7}$$

Observe that

$$\left| S_i - \bigcup_{j=1}^{i-1} S_j \right| \geq \left| S - \bigcup_{j=1}^{i-1} S_j \right| = cov_{i-1}(S) \tag{4.8}$$

because in the opposite case S would have been chosen instead of S_i in the ith run of Step 2 of Algorithm 4.3.2.11. Following (4.7) and (4.8) we obtain

$$\sum_{x \in S} Weight_C(x) \leq \sum_{i=1}^{k} (cov_{i-1}(S) - cov_i(S)) \cdot \frac{1}{\mathrm{cov}_{i-1}(S)}. \tag{4.9}$$

Since

$$Har(b) - Har(a) = \sum_{i=a+1}^{b} \frac{1}{i} \geq (b - a) \cdot \frac{1}{b}$$

we have

$$(cov_{i-1}(S) - cov_i(S)) \cdot \frac{1}{cov_{i-1}(S)} \leq Har(cov_{i-1}(S)) - Har(cov_i(S)) \tag{4.10}$$

for every $i = 1, \ldots, k$.

Inserting (4.10) into (4.9) we obtain the aimed relation (4.6):

$$\begin{aligned} \sum_{x \in S} Weight_C(x) &\leq \sum_{i=1}^{k} (Har(cov_{i-1}(S)) - Har(cov_i(S)) \\ &= Har\,(cov_0(S)) - Har(cov_k(S)) = Har(|S|) - Har(0) \\ &= Har(|S|). \end{aligned}$$

Now, we are ready to relate $|C| = cost(C) = Weight_C(C)$ to C_{Opt}. Since C_{Opt} is also a set cover of X, one has

$$|C| = \sum_{x \in X} Weight_C(x) \leq \sum_{S \in C_{Opt}} \sum_{x \in S} Weight_C(x). \tag{4.11}$$

Inserting (4.6) into (4.11) we finally obtain

$$\begin{aligned} |C| &\leq \sum_{S \in C_{Opt}} \sum_{x \in S} Weight_C(x) \leq \sum_{S \in C_{Opt}} Har(|S|) \\ &\leq |C_{Opt}| \cdot Har(\max\{|S| \mid S \in \mathcal{F}\}). \end{aligned}$$

□

Corollary 4.3.2.13 *Algorithm 4.3.2.11 is a* $\ln(|X|)$*-approximation algorithm for* SCP.

Proof. The claim follows immediately from $\max\{|S| \mid S \in \mathcal{F}\} \leq |X|$ and $Har(d) \leq \ln d$. □

Exercise 4.3.2.14 (*) Find an input instance $(X, \mathcal{F})$ for which Algorithm 4.3.2.11 provides an output with an approximation ratio in $\Omega(\ln n)$. □

Lemma 4.3.2.15 *The time complexity of Algorithm 4.3.2.11 is in* $O(n^{3/2})$.

Proof. Let us encode the inputs in such a way that one can consider $|X| \cdot |\mathcal{F}|$ to be the input size of an input instance $(X, \mathcal{F})$. One run of Step 2 costs $O(|X| \cdot |\mathcal{F}|)$ time. The number of runs of Step 2 is bounded by $\min\{|X|, |\mathcal{F}|\} \leq (|X| \cdot |\mathcal{F}|)^{1/2}$. Thus, the time complexity of Algorithm 4.3.2.11 is in $O(n^{3/2})$. □

Following Corollary 4.3.2.13 and Lemma 4.3.2.15 we obtain our main result.

Theorem 4.3.2.16 *Algorithm 4.3.2.11 is a polynomial-time* $(\ln n)$*-approximation algorithm for* SCP.[10]

Exercise 4.3.2.17 Is it possible to implement Algorithm 4.3.2.11 to compute in linear time according to $\sum_{S \in \mathcal{F}} |S|$? □

One can easily observe that MIN-VCP is a special case of SCP. For a graph $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, the corresponding input instance of SCP is $(E, \{E_1, \ldots, E_n\})$, where $E_i \subseteq E$ contains all edges adjacent to v_i for $i = 1, \ldots, n$. Algorithm 4.3.2.11, then, is a $(\ln n)$-approximation algorithm for MIN-VCP, too.

Exercise 4.3.2.18 Let $\mathcal{G}_d$ be the set of graphs with the degree bounded by d. For which d does Algorithm 4.3.2.11 ensure a better approximation ratio than Algorithm 4.3.2.1? □

[10]Note that assuming P $\neq$ NP, the approximation ratio $\ln n$ cannot be improved, so the naive greedy strategy is the best one for SCP.

Algorithm 4.3.2.1 for MIN-VCP does not provide any constant approximation ratio for WEIGHT-VCP. To design a 2-approximation algorithm for WEIGHT-VCP we use the method of relaxation by reduction to linear programming. Remember that the task is, for a given graph $G = (V, E)$ and a function $c : V \to \mathbb{N} - \{0\}$, to find a vertex cover S with the minimal cost $\sum_{v \in S} c(v)$. As already presented in Section 3.7, an instance (G, c) of WEIGHT-VCP can be expressed as the following 0/1-linear program:

$$\begin{array}{ll} \text{minimize} & \sum c(v_i) \cdot x_i \\ \text{subject to} & x_r + x_s \geq 1 \text{ for all } \{v_r, v_s\} \in E, \textit{and} \\ & x_j \in \{0,1\} \text{ for } j = 1, \ldots, n \end{array}$$

if $V = \{v_1, v_2, \ldots, v_n\}$ and $X = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$ describes the set S_X of vertices that contains v_i iff $x_i = 1$. Relaxing the constraints $x_j \in \{0,1\}$ for $j = 1, \ldots, n$ to $x_j \geq 0$ for $j = 1, 2, \ldots, n$, one obtains an instance of LP. The following algorithm solves this instance of LP and rounds the result in order to get a feasible solution to the original instance of WEIGHT-VCP.

Algorithm 4.3.2.19

Input: A graph $G = (V, E)$, and a function $c : V \to \mathbb{N} - \{0\}$. Let $V = \{v_1, v_2, \ldots, v_n\}$.

Step 1: Express the instance (G, c) as an instance $I_{0/1\text{-LP}}(G, c)$ of 0/1-linear programming and relax it to the corresponding instance $I_{\text{LP}}(G, c)$ of linear programming.

Step 2: Solve $I_{\text{LP}}(G, c)$.
Let $X = (x_1, x_2, \ldots, x_n) \in (\mathbb{R}^{\geq 0})^n$ be the computed optimal solution.

Step 3: Set $S_X := \{v_i \mid x_i \geq 1/2\}$.

Output: S_X.

Theorem 4.3.2.20 *Algorithm 4.3.2.19 is a 2-approximation algorithm for* WEIGHT-VCP.

Proof. First, we show that S_X is a feasible solution to the instance (G, c). Since X satisfies $x_r + x_s \geq 1$ for every $\{v_r, v_s\} \in E$,

$$x_r \geq 1/2 \text{ or } x_s \geq 1/2.$$

Thus, at least one of the vertices v_r and v_s is in S_X and the edge $\{v_r, v_s\}$ is covered by S_X.

Now we prove that $cost(S_X) \leq 2 \cdot Opt_{\text{WEIGHT-VCP}}(G, c)$. First of all observe that

$$Opt_{\text{WEIGHT-VCP}}(G,c) = Opt_{\text{0/1-LP}}(I_{\text{0/1-LP}}(G,c)) \geq Opt_{\text{LP}}(I_{\text{LP}}(G,c)). \quad (4.12)$$

Since

$$\begin{aligned} cost(S_X) &= \sum_{v \in S_X} c(v) = \sum_{x_i \geq 1/2} c(v_i) \leq \sum_{x_i \geq 1/2} 2 \cdot x_i \cdot c(v_i) \\ &= 2 \cdot Opt_{\text{LP}}(I_{\text{LP}}(G,c)) \underset{(4.12)}{\leq} 2 \cdot Opt_{\text{WEIGHT-VCP}}(G,c), \end{aligned}$$

the approximation ratio of Algorithm 4.3.2.19 is at most 2. □

Note that the method of relaxation to linear programming is especially suitable for the weighted versions of optimization problems. This is because of the fact that the weights does not have any influence on the number and the form of the constraints (i.e., the weighted version of a problem has the same constraints as the unweighted version of this problem), and so there is usually no essential difference between the complexity of algorithms designed by the method of relaxation to linear programming for the weighted version of an optimization problem and the simple, unweighted version of this problem.

Summary of Section 4.3.2

Very simple "heuristic" approaches like greedy or "arbitrary choice" may lead to the design of efficient approximation algorithms with a good (sometimes even best possible) approximation ratio.

MIN-VCP is a representative of the class NPO(III).

SCP is a representative of the class NPO(IV).

The method of relaxation by reduction to linear programming is often suitable for the weighted versions of optimization problems. Combining this method with rounding one can design a 2-approximation algorithm for WEIGHT-VCP.

4.3.3 Maximum Cut Problem and Local Search

The maximum cut problem (MAX-CUT) is one of the problems for which a simple local search algorithm provides outputs with a constant approximation ratio. This is due to the property that the value of any local optimum is not too far from the value of the total optima. Obviously, this property can depend on the definition of "locality" in the space of feasible solutions. For MAX-CUT it is sufficient to take a very small locality, which can be determined by the simple transformation of moving one vertex from one side of the cut to the other side of the cut.

Algorithm 4.3.3.1

Input: A graph $G = (V, E)$.

Step 1: $S = \emptyset$

{the cut is considered to be $(S, V - S)$; in fact S can be chosen arbitrarily in this step};

Step 2: **while** there exists such a vertex $v \in V$ that the movement of v from one side of the cut $(S, V - S)$ to the other side of $(S, V - S)$ increases the cost of the cut.

do begin take a $u \in V$ whose movement from one side of $(S, V - S)$ to the other side of $(S, V - S)$ increases the cost of the cut, and move this u to the other side.

end

Output: $(S, V - S)$.

Lemma 4.3.3.2 *Algorithm 4.3.3.1 runs in time* $O(|E| \cdot |V|)$.

Proof. A run of Step 2 costs $O(|V|)$ time. Step 2 can be at most $|E|$ times repeated because each run increases the cost of the current cut at least by 1 and the cost can be at most $|E|$. □

Theorem 4.3.3.3 *Algorithm 4.3.3.1 is a polynomial-time 2-approximation algorithm for* MAX-CUT.

Proof. It is obvious that Algorithm 4.3.3.1 computes a feasible solution to every given input and Lemma 4.3.3.2 proves that this happens in polynomial time.

It remains to be proven that the approximation ratio is at most 2. There is a very simple way to argue that the approximation ratio is at most 2. Let (Y_1, Y_2) be the output of Algorithm 4.3.3.1. Every vertex in Y_1 (Y_2) has at least so many edges to vertices in Y_2 (Y_1) as edges to vertices in Y_1 (Y_2). Thus, at least half of the edges of the graph is in the $cut(Y_1, Y_2)$. Since the cost of an optimal cut cannot exceed $|E|$, the proof is finished.

In what follows we give an alternative proof. The reason for giving it is to show that for many input instances the approximation ratio is clearly better than 2. Let (X_1, X_2) be an optimal cut for an input $G = (V, E)$, and let (Y_1, Y_2) be a cut computed by Algorithm 4.3.3.1, i.e., (Y_1, Y_2) is a local maximum according to the transformation used in Algorithm 4.3.3.1. Let $V_1 = Y_1 \cap X_1$, $V_2 = Y_1 \cap X_2$, $V_3 = Y_2 \cap X_1$, and $V_4 = Y_2 \cap X_2$ (see Figure 4.4).

Thus, $(X_1, X_2) = (V_1 \cup V_3, V_2 \cup V_4)$, and $(Y_1, Y_2) = (V_1 \cup V_2, V_3 \cup V_4)$ because $X_1 \cup X_2 = Y_1 \cup Y_2 = V$. Let $e_{ij} = |E \cap \{\{x, y\} \mid x \in V_i, y \in V_j\}|$ be the number of edges between V_i and V_j for $1 \leq i \leq j \leq 4$. Thus,

$$cost(X_1, X_2) = e_{12} + e_{14} + e_{23} + e_{34},$$

and

$$cost(Y_1, Y_2) = e_{13} + e_{14} + e_{23} + e_{24}.$$

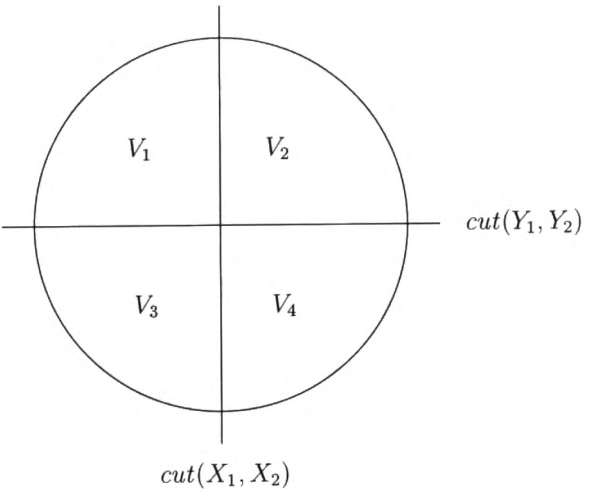

Fig. 4.4.

Our aim is to show

$$cost(X_1, X_2) \leq 2 \cdot cost(Y_1, Y_2) - \left(\sum_{i=1}^{4} e_{ii} + e_{13} + e_{24} \right).$$

For every vertex x in V_1, the number of edges between x and the vertices in $V_1 \cup V_2$ is at most the number of edges between x and the vertices in $V_3 \cup V_4$. (In the opposite case, Algorithm 4.3.3.1 would move x from $Y_1 = V_1 \cup V_2$ to $Y_2 = V_3 \cup V_4$). Summing via all vertices in V_1 we obtain

$$2e_{11} + e_{12} \leq e_{13} + e_{14}. \tag{4.13}$$

Because none of the cuts (X_1, X_2) and (Y_1, Y_2) can be improved by the local transformation of Algorithm 4.3.3.1, analogous claims as those for V_1 are true for every vertex of V_2, V_3, and V_4, respectively. This results in the following inequalities:

$$2e_{22} + e_{12} \leq e_{23} + e_{24} \tag{4.14}$$
$$2e_{33} + e_{34} \leq e_{23} + e_{13} \tag{4.15}$$
$$2e_{44} + e_{34} \leq e_{14} + e_{24}. \tag{4.16}$$

Taking ((4.13) + (4.14) + (4.15) + (4.16))/2 one obtains

$$\sum_{i=1}^{4} e_{ii} + e_{12} + e_{34} \leq e_{13} + e_{14} + e_{23} + e_{24}. \tag{4.17}$$

Adding $e_{14} + e_{23}$ to both sides of the relation (4.17) we get

$$\sum_{i=1}^{4} e_{ii} + e_{12} + e_{34} + e_{14} + e_{23} \leq 2e_{23} + 2e_{14} + e_{13} + e_{24}. \tag{4.18}$$

Now we obtain

$$\begin{aligned} cost(X_1, X_2) &= e_{12} + e_{34} + e_{14} + e_{23} \\ &\leq \sum_{i=1}^{4} e_{ii} + e_{12} + e_{14} + e_{34} + e_{23} \\ &\underset{(4.18)}{\leq} 2e_{23} + 2e_{14} + e_{13} + e_{24} \\ &\leq 2(e_{23} + e_{14} + e_{13} + e_{24}) = 2 \cdot cost(Y_1, Y_2) \end{aligned}$$

The above calculation directly implies

$$cost(X_1, X_2) \leq 2 \cdot cost(Y_1, Y_2) - \sum_{i=1}^{4} e_{ii} - e_{13} - e_{24}.$$

□

Exercise 4.3.3.4 Show that one can even prove $cost(X_1, X_2) < 2 \cdot cost(Y_1, Y_2)$ in the proof of Theorem 4.3.3.3. □

Exercise 4.3.3.5 Find input instances for which the approximation ratio of the solutions computed by Algorithm 4.3.3.1 is almost 2. □

Exercise 4.3.3.6 Let k-maximum cut problem (k-MAX-CUT) be a restricted version of MAX-CUT, where the degree of the input graphs is bounded by $k \geq 3$. Design a δ_k-approximation algorithm for k-MAX-CUT for every $k \geq 3$ in such a way that $\delta_k < \delta_{k+1} < 2$ for every $k \geq 3$. □

Summary of Section 4.3.3

There are hard optimization problems whose local optima (according to a reasonable structure of the space of feasible solutions) have costs that well approximate the cost of total optima. In such cases local search algorithms may provide a very efficient solution to the given optimization problem.

4.3.4 Knapsack Problem and PTAS

The knapsack problem (KP) is a representative of the class NPO(I) and so KP is one of the easiest NP-hard optimization problems according to the polynomial-time approximability. We use this problem to illustrate a few concepts for the