Approximation Schemes – A Tutorial

Petra Schuurman [1] Gerhard J. Woeginger [2]

[1] petra@win.tue.nl. Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.

[2] gwoegi@opt.math.tu-graz.ac.at. Institut für Mathematik B, TU Graz, Steyrergasse 30, A-8010 Graz, Austria.

Summary: This tutorial provides an introduction into the area of polynomial time approximation schemes. The underlying ideas, the main tools, and the standard approaches for the construction of such schemes are explained in great detail and illustrated in many examples and exercises. The tutorial also discusses techniques for disproving the existence of approximation schemes.

Keywords: approximation algorithm, approximation scheme, PTAS, FPTAS, worst case analysis, performance guarantee, in-approximability, gap technique, L-reduction, APX, intractability, polynomial time, scheduling.

0.1 Introduction

All interesting problems are difficult to solve. This observation holds especially in algorithm oriented areas (like combinatorial optimization, mathematical programming, operations research, or theoretical computer science) where researchers often face computationally intractable problems. Since solving an intractable problem to optimality is a tough thing to do, these researchers usually resort to simpler suboptimal approaches that yield decent solutions, and hope that those decent solutions come at least close to the true optimum. An approximation scheme is a suboptimal approach that provably works fast and that provably yields solutions of very high quality. That is the topic of this chapter: Approximation schemes. Let us illustrate this concept by an example taken from the real world of Dilbert [1] cartoons.

Example 0.1.1 *Suppose that you are the pointy-haired boss of the planning department of a huge company. The top-managers of your company have decided that the company is going to make a gigantic profit by constructing a spaceship that travels faster than light. Your department has to set up the schedule for this project. And of course, the top-managers want you to find a schedule that minimizes the cost incurred by the company. So, you ask your experienced programmers Wally and Dilbert to determine such a schedule. Wally and Dilbert tell you: "We can do that. Real programmers can do everything. And we guess that the cost of the best schedule will be exactly one zillion dollars." You say: "Sounds great! Wonderful! Go ahead and determine this schedule! I will present it to the top-managers in the meeting tomorrow afternoon." Then Wally and Dilbert say: "We cannot do that by tomorrow afternoon. Real programmers can do everything, but finding the schedule is going to take us twenty-three and a half years."*

You are deeply shocked by the incompetence of your employees, and you decide to give the task to somebody who is really smart. So, you consult Dogbert, the dog of Dilbert. Dogbert tells you: "Call me up tomorrow, and I will have your schedule. The schedule is going to cost exactly two zillion dollars." You complain: "But Wally and Dilbert told me that there is a schedule that only costs one zillion dollars. I do not want to spend an additional zillion dollars on it!" And Dogbert says: "Then please call me up again twenty-three and a half years *from now. Or, you may call me up again the day after tomorrow, and I will have a schedule for you that only costs one and a half zillion dollars. Or, you may call me up again the day after the day after tomorrow, and I will have a schedule that only costs one and a third zillion dollars." Now you become really curious: "What if I call you up exactly x days from now?" Dogbert: "Then I would have found a schedule that costs at most $1 + 1/x$ zillion dollars."*

Dogbert obviously has found an approximation scheme for the company's tough scheduling problem: Within reasonable time (which means: x days) he can come fairly close (which means: at most a factor of $1+1/x$ away) to the true optimum (which means: one zillion dollars). Note that as x becomes very large, the cost of Dogbert's schedule comes arbitrarily close to the optimal cost. The goal of this chapter is to give you a better understanding of Dogbert's technique. We will introduce you to the main ideas, and we will explain the standard tools for finding approximation schemes. We identify three constructive approaches for getting an approximation scheme, and we illustrate their underlying ideas by stating many examples and exercises. Of course, not every optimization problem has an approximation scheme – this just would be too good to be true. We will explain how one can recognize optimization problems with bad approximability behavior. Currently there are only a few tools available for getting such in-approximability results, and we will discuss them in detail and illustrate them with many examples.

The chapter uses the context of scheduling to present the techniques and tools around approximation schemes, and all the illustrating examples and exercises are taken from the field of scheduling. However, the methodology is general and it applies to all kinds of optimization problems in all kinds of areas like networks, graph theory, geometry, etc.

———————————

In the following paragraphs we will give exact mathematical definitions of the main concepts in the area of approximability. For these paragraphs and also for the rest of the chapter, we will assume that the reader is familiar with the basic concepts in computational complexity theory that are listed in the Appendix Section 0.8.

An optimization problem is specified by a set $\mathcal{I}$ of inputs (or instances), by a set $\mathrm{SOL}(I)$ of feasible solutions for every input $I \in \mathcal{I}$, and by an objective function c that specifies for every feasible solution σ in $\mathrm{SOL}(I)$ an objective value or cost $c(\sigma)$. We will only consider optimization problems in which all feasible solutions have non-negative cost. An optimization problem may either be a minimization problem where the optimal solution is a feasible solution with minimum possible cost, or a maximization problem where the optimal solution is a feasible solution with maximum possible cost. In any case, we will denote the optimal objective value for instance I by $\mathrm{OPT}(I)$. By $|I|$ we denote the size of an instance I, i.e., the number of bits used in writing down I in some fixed encoding. Now assume that we are dealing with an NP-hard optimization problem where it is difficult to find the exact optimal solution within polynomial time in $|I|$. At the expense of reducing the quality of the solution, we can often get considerable speed-up in the time complexity. This motivates the following definition.

Definition 0.1.2 *(Approximation algorithms)*
Let X be a minimization (respectively, maximization) problem. Let $\varepsilon > 0$, and set $\rho = 1+\varepsilon$ (respectively, $\rho = 1-\varepsilon$). An algorithm A is called a ρ-approximation algorithm for problem X, if for all instances I of X it delivers a feasible solution with objective value $A(I)$ such that

$$|A(I) - \textsc{Opt}(I)| \;\leq\; \varepsilon \cdot \textsc{Opt}(I). \tag{0.1}$$

In this case, the value ρ is called the performance guarantee or the worst case ratio of the approximation algorithm A.

Note that for minimization problems the inequality in (0.1) becomes $A(I) \leq (1+\varepsilon)\textsc{Opt}(I)$, whereas for maximization problems it becomes $A(I) \geq (1-\varepsilon)\textsc{Opt}(I)$. Note furthermore that for minimization problems the worst case ratio $\rho = 1+\varepsilon$ is a real number greater or equal to 1, whereas for maximization problems the worst case ratio $\rho = 1-\varepsilon$ is a real number from the interval $[0,1]$. The value ρ can be viewed as the quality measure of the approximation algorithm. The closer ρ is to 1, the better the algorithm is. A worst case ratio $\rho = 0$ for a maximization problem, or a worst case ratio $\rho = 10^6$ for a minimization problem are of rather poor quality. The complexity class APX consists of all minimization problems that have a polynomial time approximation algorithm with some finite worst case ratio, and of all maximization problems that have a polynomial time approximation algorithm with some positive worst case ratio.

Definition 0.1.3 *(Approximation schemes)*
Let X be a minimization (respectively, maximization) problem.

- *An approximation scheme for problem X is a family of $(1+\varepsilon)$-approximation algorithms A_ε (respectively, $(1-\varepsilon)$-approximation algorithms A_ε) for problem X over all $0 < \varepsilon < 1$.*

- *A polynomial time approximation scheme (PTAS) for problem X is an approximation scheme whose time complexity is polynomial in the input size.*

- *A fully polynomial time approximation scheme (FPTAS) for problem X is an approximation scheme whose time complexity is polynomial in the input size and also polynomial in $1/\varepsilon$.*

Hence, for a PTAS it would be acceptable to have a time complexity proportional to $|I|^{2/\varepsilon}$; although this time complexity is exponential in $1/\varepsilon$, it is polynomial in the size of the input I exactly as we required in the definition of a PTAS. An FPTAS cannot have a time complexity that grows exponentially in $1/\varepsilon$, but a time complexity proportional to $|I|^8/\varepsilon^3$ would be fine. With respect to worst case approximation, an FPTAS is the strongest possible result that we can derive for an NP-hard problem. Figure 0.1 illustrates the relationships between the classes NP, APX, P, the class of problems that are pseudo-polynomially solvable, and the classes of problems that have a PTAS and FPTAS.

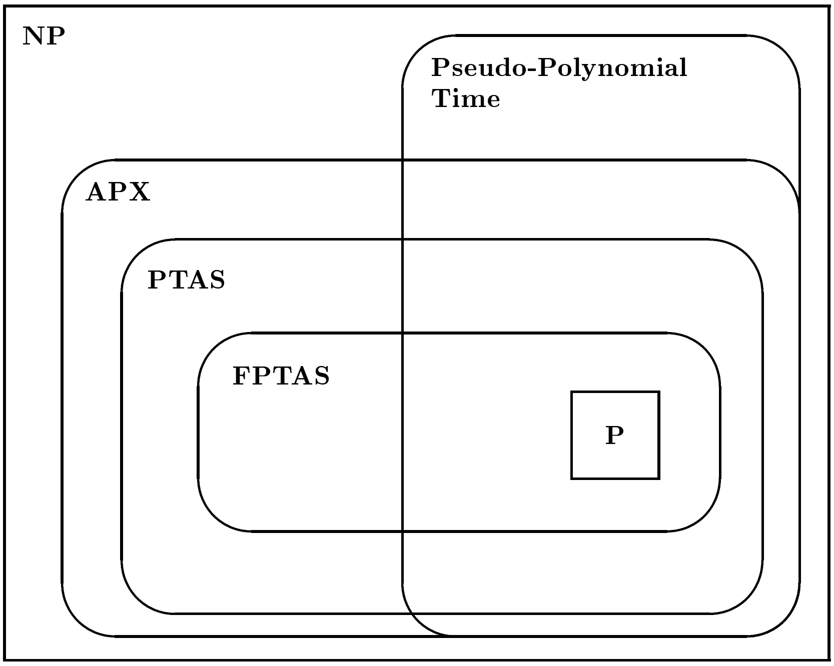

Figure 0.1: Relationships between some of the complexity classes discussed in this chapter.

A little bit of history. The first paper with a polynomial time approximation algorithm for an NP-hard problem is probably the paper [26] by Graham from 1966. It studies simple heuristics for scheduling on identical parallel machines. In 1969, Graham [27] extended his approach to a PTAS. However, at that time these were isolated results. The concept of an approximation algorithm was formalized in the beginning of the 1970s by Garey, Graham & Ullman [21]. The paper [45] by Johnson may be regarded as the real starting point of the field; it raises the 'right' questions on the approximability of a wide range of optimization problems. In the mid-1970s, a number of PTAS's was developed in the work of Horowitz & Sahni [42, 43], Sahni [69, 70], and Ibarra & Kim [44].

The terms 'approximation scheme', 'PTAS', 'FPTAS' are due to a seminal paper by Garey & Johnson [23] from 1978. Also the first in-approximability results were derived around this time; in-approximability results are results that show that unless P=NP some optimization problem does not have a PTAS or that some optimization problem does not have a polynomial time ρ-approximation algorithm for some specific value of ρ. Sahni & Gonzalez [71] proved that the traveling salesman problem without the triangle-inequality cannot have a polynomial time approximation algorithm with finite worst case ratio. Garey & Johnson [22] derived in-approximability results for the chromatic number of a graph. Lenstra & Rinnooy Kan [58] derived in-approximability results for scheduling of precedence constrained jobs.

In the 1980s theoretical computer scientists started a systematic theoretical study of these concepts; see for instance the papers Ausiello, D'Atri & Protasi [11], Ausiello, Marchetti-Spaccamela & Protasi [12], Paz & Moran [65], and Ausiello, Crescenzi & Protasi [10]. They derived deep and beautiful characterizations of polynomial time approximable problems. These theoretical characterizations are usually based on the existence of certain polynomial time computable functions that are related to the optimization problem in a certain way, and the characterizations do not provide any help in identifying these functions and in constructing the PTAS. The reason for this is of course that all these characterizations implicitly suffer from the difficulty of the P=NP question.

Major breakthroughs that give PTAS's for specific optimization problems were the papers by Fernandez de la Vega & Lueker [20] on bin packing, by Hochbaum & Shmoys [37, 38] on the scheduling problem of minimizing the makespan on an arbitrary number of parallel machines, by Baker [14] on many optimization problems on planar graphs (like maximum independent set, minimum vertex cover, minimum dominating set), and by Arora [5] on the Euclidean traveling salesman problem. In the beginning of the 1990s, Papadimitriou & Yannakakis [64] provided tools and ideas from computational complexity theory for getting in-approximability results. The complexity class APX was born; this class contains all optimization problems that possess a polynomial time approximation algorithm with a finite, positive worst case ratio. In 1992 Arora, Lund, Motwani, Sudan & Szegedy [6] showed that the hardest problems in APX cannot have a PTAS unless P=NP. For an account of the developments that led to these in-approximability results see the NP-completeness column [46] by Johnson.

Organization of this chapter. Throughout the chapter we will distinguish between so-called *positive results* which establish the existence of some approximation scheme, and so-called *negative results* which disprove the existence of good approximation results for some optimization problem under the assumption that P≠NP. Sections 0.2–0.5 are on positive results. First, in Section 0.2 we introduce three general approaches for the construction of approximation schemes. These three approaches are then analyzed in detail and illustrated with many examples in the subsequent three Sections 0.3, 0.4, and 0.5. In Section 0.6 we move on to methods for deriving negative results. At the end of each of the Sections 0.3–0.6 there are lists of exercises. Section 0.7 contains a brief conclusion. Finally, the Appendix Section 0.8 gives a very terse introduction into computational complexity theory.

Some remarks on the notation. Throughout the chapter, we use the standard three-field $\alpha \,|\, \beta \,|\, \gamma$ scheduling notation (see for instance Graham, Lawler, Lenstra & Rinnooy Kan [28] and Lawler, Lenstra, Rinnooy Kan & Shmoys [55]). The field α specifies the machine environment, the field β specifies the job environment, and the field γ specifies the objective function.

We denote the base two logarithm of a real number x by $\log(x)$, its natural logarithm by $\ln(x)$, and its base b logarithm by $\log_b(x)$. For a real number x, we denote by $\lfloor x \rfloor$ the largest integer less or equal to x, and we denote by $\lceil x \rceil$ the smallest integer greater or equal to x. Note that $\lceil x \rceil + \lceil y \rceil \geq \lfloor x + y \rfloor$ and $\lfloor x \rfloor + \lfloor y \rfloor \leq \lceil x + y \rceil$ hold for all real numbers x and y. A d-dimensional vector $\vec{v}$ with coordinates v_k $(1 \leq k \leq d)$ will always be written in square brackets as $\vec{v} = [v_1, v_2, \ldots, v_d]$. For two d-dimensional vectors $\vec{v} = [v_1, v_2, \ldots, v_d]$ and $\vec{u} = [u_1, u_2, \ldots, u_d]$ we write $\vec{u} \leq \vec{v}$ if and only if $u_k \leq v_k$ holds for $1 \leq k \leq d$. For a finite set S, we denote its cardinality by $|S|$. For an instance I of a computational problem, we denote its size by $|I|$, i.e., the number of bits that are used for writing down I in some fixed encoding.

0.2 How to get positive results

Positive results in the area of approximation concern the design and analysis of polynomial time approximation algorithms and polynomial time approximation schemes. This section (and also the following three sections of this chapter) concentrate on such positive results; in this section we will only outline the main strategy. Assume that we need to find an approximation scheme for some fixed NP-hard optimization problem X. How shall we proceed?

Let us start by considering an exact algorithm A that solves problem X to optimality. Algorithm A takes an instance I of X, processes it for some time, and finally outputs the solution $A(I)$ for instance I. See Figure 0.2 for an illustration. All known approaches to approximation schemes are based on the diagram depicted in this figure. Since the optimization problem X is difficult to solve, the exact algorithm A will have a bad (exponential) time complexity and will be far away from yielding a PTAS or yielding an FPTAS. How can we improve the behavior of such an algorithm and bring it closer to a PTAS? The answer is to add *structure* to the diagram in Figure 0.2. This additional structure depends on the desired precision ε of approximation. If ε is large, there should be lots of additional structure. And as ε tends to 0, also the amount of additional structure should tend to 0 and should eventually disappear. The additional structure simplifies the situation and leads to simpler, perturbed and blurred versions of the diagram in Figure 0.2.

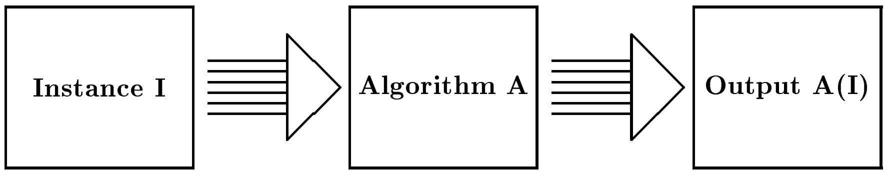

Figure 0.2: Algorithm A solves instance I and outputs the feasible solution $A(I)$.

Note that the diagram consists of three well-separated parts: The input to the left, the output to the right, and the execution of the algorithm A in the middle. And these three well-separated parts give us three ways to add structure to the diagram. The three ways will be discussed in the following three sections: Section 0.3 deals with the addition of structure to the input of an algorithm, Section 0.4 deals with the addition of structure to the output of an algorithm, and Section 0.5 deals with the addition of structure to the execution of an algorithm.

0.3 Structuring the input

As first standard approach to the construction of approximation schemes we will discuss the technique of *adding structure to the input data*. Here the main idea is to turn a difficult instance into a more primitive instance that is easier to tackle. Then we use the optimal solution for the primitive instance to get a grip on the original instance. More formally, the approach can be described by the following three-step procedure; see Figure 0.3 for an illustration.

(A) Simplify. Simplify instance I into a more primitive instance $I^{\#}$. This simplification depends on the desired precision ε of approximation; the closer ε is to zero, the closer instance $I^{\#}$ should resemble instance I. The time needed for the simplification must be polynomial in the input size.

(B) Solve. Determine an optimal solution $\textsc{Opt}^{\#}$ for the simplified instance $I^{\#}$ in polynomial time.

(C) Translate back. Translate the solution $\textsc{Opt}^{\#}$ for $I^{\#}$ back into an approximate solution $\textsc{App}$ for instance I. This translation exploits the similarity between instances I and $I^{\#}$. In the ideal case, $\textsc{App}$ will stay close to $\textsc{Opt}^{\#}$ which in turn is close to $\textsc{Opt}$. In this case we find an excellent approximation.

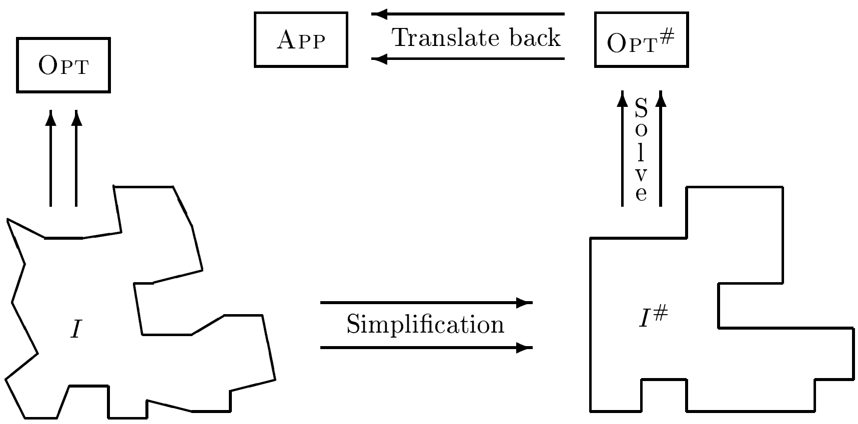

Figure 0.3: Structuring the input. Instance I is very complicated and irregularly shaped, and it would be difficult to go directly from I to its optimal solution Opt. Hence, one takes the detour via the simplified instance $I^{\#}$ for which it is easy to obtain an optimal solution Opt$^{\#}$. Then one translates Opt$^{\#}$ into an approximate solution App for the original instance I. Let us hope that the objective value of App is close to that of Opt!

Of course, finding the right simplification in step (A) is an art. If instance $I^{\#}$ is chosen too close to the original instance I, then $I^{\#}$ might still be NP-hard to solve to optimality. On the other hand, if instance $I^{\#}$ is chosen too far away from the original instance I, then solving $I^{\#}$ will not tell us anything about how to solve I. Under-simplifications (for instance, setting $I^{\#} = I$) and over-simplifications (for instance, setting $I^{\#} = \emptyset$) are equally dangerous. The following approaches to simplifying the input often work well.

Rounding. The simplest way of adding structure to the input is to *round* some of the numbers in the input. For instance, we may round all job lengths to perfect powers of two, or we may round non-integral due dates up to the closest integers.

Merging. Another way of adding structure is to *merge* small pieces into larger pieces of primitive shape. For instance, we may merge a huge number of tiny jobs into a single job with processing time equal to the processing time of the tiny jobs, or into a single job with processing time equal to the processing time of the tiny jobs rounded to some nice value.

Cutting. Yet another way of adding structure is to cut away irregular shaped pieces from the instance. For instance, we may remove a small set of jobs with a broad spectrum of processing times from the instance.

Aligning. Another way of adding structure to the input is to *align* the shapes of several similar items. For instance, we may replace thirty-six different jobs of roughly equal length by thirty-six identical copies of the job with median length.

The approach of structuring the input has a long history that goes back (at least) to the early 1970s. In 1974 Horowitz & Sahni [42] used it to attack partition problems. In 1975 Sahni [69] applied it to the 0-1 knapsack problem, and in 1976 Sahni [70] applied it to makespan minimization on two parallel machines. Other prominent approximation schemes that use this approach can be found in the paper of Fernandez de la Vega & Lueker [20] on bin packing, and in the paper by Hochbaum & Shmoys [37] on makespan minimization on parallel machines (the Hochbaum & Shmoys result will be discussed in detail in Section 0.3.2). More recently, Arora [5] applied simplification of the input as a kind of preprocessing step in his PTAS for the Euclidean traveling salesman problem, and Van Hoesel & Wagelmans [77] used it to develop an FPTAS for the economic lot-sizing problem.

In the following three sections, we will illustrate the technique of simplifying the input data with the help of three examples. Section 0.3.1 deals with makespan minimization on two identical machines, Section 0.3.2 deals with makespan minimization on an arbitrary number of identical machines, and Section 0.3.3 discusses total tardiness on a single machine. Section 0.3.4 contains a number of exercises.

0.3.1 Makespan on two identical machines

The problem. In the scheduling problem $P2\,|\,|\,C_{\max}$ the input consists of n jobs J_j ($j = 1, \ldots, n$) with positive integer processing times p_j. All jobs are available at time zero, and preemption is not allowed. The goal is to schedule the jobs on two identical parallel machines so as to minimize the maximum job completion time, the so-called *makespan* $C_{\max}$. In other words, we would like to assign roughly equal amounts of processing time to both machines (but without cutting any of the jobs); the objective value is the total processing time on the machine that finishes last. Throughout this section (and also in all subsequent sections), the objective value of an optimal schedule will be denoted by $\textsc{Opt}$.

The problem $P2\,|\,|\,C_{\max}$ is NP-hard in the ordinary sense (Karp [48]). We denote by $p_{\text{sum}} = \sum_{j=1}^{n} p_j$ the overall job processing time and by $p_{\max} = \max_{j=1}^{n} p_j$ the length of the longest job. It is easy to see that for $L = \max\{\frac{1}{2}p_{\text{sum}}, p_{\max}\}$

$$L \;\le\; \textsc{Opt}. \tag{0.2}$$

Indeed, $p_{\max}$ is a lower bound on $\textsc{Opt}$ (the longest job must be entirely processed on one of the two machines) and also $\frac{1}{2}p_{\text{sum}}$ is a lower bound on $\textsc{Opt}$ (the overall job processing time p_{sum} must be assigned to the two machines, and even if we reach a perfect split the makespan is still at least $\frac{1}{2}p_{\text{sum}}$).

In this section, we will construct a PTAS for problem $P2\,|\,|\,C_{\max}$ by applying the technique of simplifying the input. Later in this chapter we will meet this problem again, once in Section 0.4.1 and once in Section 0.5.1: Since $P2\,|\,|\,C_{\max}$ is very simple to state and since it is a very basic problem in scheduling, it is the perfect candidate to illustrate all the three main approaches to approximation schemes. As we will see, the three resulting approximation schemes are completely independent and very different from each other.

(A) How to simplify an instance. We translate an arbitrary instance I of $P2\,|\,|\,C_{\max}$ into a corresponding simplified instance $I^{\#}$. The jobs in I are classified into *big jobs* and *small jobs*; the classification depends on a precision parameter $0 < \varepsilon < 1$.

- A job J_j is called *big* if it has processing time $p_j > \varepsilon L$. The instance $I^{\#}$ contains all the big jobs from instance I.
- A job J_j is called *small* if it has processing time $p_j \leq \varepsilon L$. Let S denote the total processing time of all small jobs in I. Then instance $I^{\#}$ contains $\lfloor S/(\varepsilon L)\rfloor$ jobs of length εL. In a pictorial setting, the small jobs in I are first glued together to give a long job of length S, and then this long job is cut into lots of chunks of length εL (if the last chunk is strictly smaller than εL, then we simply disregard it).

And this completes the description of the simplified instance $I^{\#}$. Why do we claim that $I^{\#}$ is a simplified version of instance I? Well, the big jobs in I are copied directly into $I^{\#}$. For the small jobs in I, we imagine that they are like sand; their exact size does not matter, but we must be able to fit all this sand into the schedule. Since the chunks of length εL in $I^{\#}$ cover about the same space as the small jobs in I do, the most important properties of the small jobs are also present in instance $I^{\#}$.

We want to argue that the optimal makespan $\textsc{Opt}^{\#}$ of $I^{\#}$ is fairly close to the optimal makespan $\textsc{Opt}$ of I: Denote by S_i $(1 \leq i \leq 2)$ the total size of all small jobs on machine M_i in an optimal schedule for I. On M_i, leave every big job where it is, and replace the small jobs by $\lceil S_i/(\varepsilon L)\rceil$ chunks of length εL. Since

$$\lceil S_1/(\varepsilon L)\rceil + \lceil S_2/(\varepsilon L)\rceil \;\geq\; \lfloor S_1/(\varepsilon L) + S_2/(\varepsilon L)\rfloor \;=\; \lfloor S/(\varepsilon L)\rfloor,$$

this process assigns all the chunks of length εL to some machine. By assigning the chunks, we increase the load of M_i by at most $\lceil S_i/(\varepsilon L)\rceil\varepsilon L - S_i \leq (S_i/(\varepsilon L) + 1)\,\varepsilon L - S_i = \varepsilon L$. The resulting schedule is a feasible schedule for instance $I^{\#}$. We conclude that

$$\textsc{Opt}^{\#} \;\leq\; \textsc{Opt} + \varepsilon L \;\leq\; (1+\varepsilon)\textsc{Opt}. \tag{0.3}$$

Note that the stronger inequality $\textsc{Opt}^{\#} \leq \textsc{Opt}$ will not in general hold. Consider for example an instance that consists of six jobs of length 1 with $\varepsilon = 2/3$.

Then $\text{OPT} = L = 3$, and all the jobs are small. In $I^{\#}$ they are replaced by 3 chunks of length 2, and this leads to $\text{OPT}^{\#} = 4 > \text{OPT}$.

(B) How to solve the simplified instance. How many jobs are there in instance $I^{\#}$? When we replaced the small jobs in instance I by the chunks in instance $I^{\#}$, we did not increase the total processing time. Hence, the total processing time of all jobs in $I^{\#}$ is at most $p_{\text{sum}} \le 2L$. Since each job in $I^{\#}$ has length at least εL, there are at most $2L/(\varepsilon L) = 2/\varepsilon$ jobs in instance $I^{\#}$. The number of jobs in $I^{\#}$ is bounded by a finite constant that only depends on ε and thus is completely independent of the number n of jobs in I.

Solving instance $I^{\#}$ is easy as pie! We may simply try all possible schedules! Since each of the $2/\varepsilon$ jobs is assigned to one of the two machines, there are at most $2^{2/\varepsilon}$ possible schedules, and the makespan of each of these schedules can be determined in $O(2/\varepsilon)$ time. So, instance $I^{\#}$ can be solved in constant time! Of course this 'constant' is huge and grows exponentially in $1/\varepsilon$, but after all our goal is to get a PTAS (and not an FPTAS), and so we do not care at all about the dependence of the time complexity on $1/\varepsilon$.

(C) How to translate the solution back. Consider an optimal schedule $\sigma^{\#}$ for the simplified instance $I^{\#}$. For $i = 1, 2$ we denote by $L_i^{\#}$ the load of machine M_i in this optimal schedule, by $B_i^{\#}$ the total size of the big jobs on M_i, and by $S_i^{\#}$ the total size of the chunks of small jobs on M_i. Clearly, $L_i^{\#} = B_i^{\#} + S_i^{\#}$ and

$$S_1^{\#} + S_2^{\#} \;=\; \varepsilon L \cdot \lfloor \frac{S}{\varepsilon L} \rfloor \;>\; S - \varepsilon L. \tag{0.4}$$

We construct the following schedule σ for I: Every big job is put onto the same machine as in schedule $\sigma^{\#}$. How shall we handle the small jobs? We reserve an interval of length $S_1^{\#} + 2\varepsilon L$ on machine M_1, and an interval of length $S_2^{\#}$ on machine M_2. We then greedily put the small jobs into these reserved intervals: First, we start packing small jobs into the reserved interval on M_1, until we meet some small job that does not fit in any more. Since the size of a small job is at most εL, the total size of the packed small jobs on M_1 is at least $S_1^{\#} + \varepsilon L$. Then the total size of the unpacked jobs is at most $S - S_1^{\#} - \varepsilon L$, and by (0.4) this is bounded from above by $S_2^{\#}$. Hence, all remaining unpacked small jobs together will fit into the reserved interval on machine M_2. This completes the description of schedule σ for instance I.

Let us compare the loads L_1 and L_2 of the machines in σ to the machine completion times $L_1^{\#}$ and $L_2^{\#}$ in schedule $\sigma^{\#}$. Since the total size of the small jobs on M_i is at most $S_i^{\#} + 2\varepsilon L$, we conclude that

$$L_i \;\le\; B_i^{\#} + (S_i^{\#} + 2\varepsilon L) \;=\; L_i^{\#} + 2\varepsilon L \;\le\; (1+\varepsilon)\text{OPT} + 2\varepsilon\text{OPT} \;=\; (1+3\varepsilon)\text{OPT}. \tag{0.5}$$

In this chain of inequalities we have used $L \le \text{OPT}$ from (0.2), and $L_i^{\#} \le \text{OPT}^{\#} \le (1+\varepsilon)\text{OPT}$ which follows from (0.3). Hence, the makespan of schedule

σ is at most a factor $1+3\varepsilon$ above the optimum makespan. Since 3ε can be made arbitrary close to 0, we finally have reached the desired PTAS for $P2\,|\,|\,C_{\max}$.

Discussion. At this moment, the reader might wonder whether for designing the PTAS, it is essential whether we round the numbers up or whether we do round them down. If we are dealing with the makespan criterion, then it usually is not essential. If the rounding is done in a slightly different way, all our claims and all the used inequalities still hold in some slightly modified form. For example, suppose that we had defined the number of chunks in instance $I^{\#}$ to be $\lceil S/(\varepsilon L)\rceil$ instead of $\lfloor S/(\varepsilon L)\rfloor$. Then inequality (0.3) could be replaced by $\text{OPT}^{\#} \le (1+2\varepsilon)\text{OPT}$. All our calculations could by updated in an appropriate way, and eventually they would yield a worst case guarantee of $1+4\varepsilon$. This again yields a PTAS. Hence, there is lots of leeway in our argument and there is sufficient leeway to do the rounding in a different way.

The time complexity of our PTAS is linear in n, but exponential in $1/\varepsilon$: The instance $I^{\#}$ is easily determined in $O(n)$ time, the time for solving $I^{\#}$ grows exponentially with $1/\varepsilon$, and translating the solution back can again be done in $O(n)$ time. Is this the best time complexity we can expect from a PTAS for $P2\,|\,|\,C_{\max}$? No, there is even an FPTAS (but we will have to wait till Section 0.5.1 to see it).

How would we tackle makespan minimization on $m \ge 3$ machines? First, we should redefine $L = \max\{\frac{1}{m}p_{\text{sum}}, p_{\max}\}$ instead of $L = \max\{\frac{1}{2}p_{\text{sum}}, p_{\max}\}$ so that the crucial inequality (0.2) is again satisfied. The simplification step (A) and the translation step (C) do not need major modifications. Exercise 0.3.1 in Section 0.3.4 asks the reader to fill in the necessary details. However, the simplified instance $I^{\#}$ in step (B) now may consist of roughly m/ε jobs, and so the time complexity becomes exponential in m/ε. As long as the number m of machines is a fixed constant, this approach works fine and gives us a PTAS for $Pm\,|\,|\,C_{\max}$. But if m is part of the input, the approach breaks down. The corresponding problem $P\,|\,|\,C_{\max}$ is the subject of the following section.

0.3.2 Makespan on an arbitrary number of identical machines

The problem. In the scheduling problem $P\,|\,|\,C_{\max}$ the input consists of n jobs J_j ($j = 1, \ldots, n$) with positive integer processing times p_j and of m identical machines. The goal is to find a schedule that minimizes the makespan; the optimal makespan will be denoted by OPT. This problem generalizes $P2\,|\,|\,C_{\max}$ from Section 0.3.1. The crucial difference to $P2\,|\,|\,C_{\max}$ is that the number m of machines is part of the input. Therefore, a polynomial time algorithm dealing with this problem must have a time complexity that is also polynomially bounded in m.

The problem $P\,|\,|\,C_{\max}$ is NP-hard in the strong sense (Garey & Johnson [24]). Analogously to Section 0.3.1, we define $p_{\text{sum}} = \sum_{j=1}^{n} p_j$, $p_{\max} =$

$\max_{j=1}^{n} p_j$, and $L = \max\{\frac{1}{m}p_{\rm sum}, p_{\rm max}\}$. We claim that

$$L \;\le\; \textsc{Opt} \;\le\; 2L. \tag{0.6}$$

Indeed, since $\frac{1}{m}p_{\rm sum}$ and $p_{\rm max}$ are lower bounds on $\textsc{Opt}$, L is also a lower bound on $\textsc{Opt}$. We show that $2L$ is an upper bound on $\textsc{Opt}$ by exhibiting a schedule with makespan at most $2L$. We assign the jobs one by one to the machines; every time a job is assigned, it is put on a machine with the current minimal workload. As this minimal workload is at most $\frac{1}{m}p_{\rm sum}$ and as the newly assigned job adds at most $p_{\rm max}$ to the workload, the makespan always remains bounded by $\frac{1}{m}p_{\rm sum} + p_{\rm max} \le 2L$.

We will now construct a PTAS for problem $P\,||\,C_{\rm max}$ by the technique of simplifying the input. Many ideas and arguments from Section 0.3.1 directly carry over to $P\,||\,C_{\rm max}$. We mainly concentrate on the additional ideas that were developed by Hochbaum & Shmoys [37] and by Alon, Azar, Woeginger & Yadid [3] to solve the simplified instance in step (B).

(A) How to simplify an instance. To simplify the presentation, we assume that $\varepsilon = 1/E$ for some integer E. Jobs are classified into big and small ones, exactly as in Section 0.3.1. The small jobs with total size S are replaced be $\lfloor S/(\varepsilon L)\rfloor$ chunks of length εL, exactly as in Section 0.3.1. The big jobs, however, are handled in a different way: For each big job J_j in I, the instance $I^\#$ contains a corresponding job $J_j^\#$ with processing time $p_j^\# = \varepsilon^2 L\lfloor p_j/(\varepsilon^2 L)\rfloor$, i.e., $p_j^\#$ is obtained by rounding p_j down to the next integer multiple of $\varepsilon^2 L$. Note that $p_j \le p_j^\# + \varepsilon^2 L \le (1+\varepsilon)p_j^\#$ holds. This yields the simplified instance $I^\#$.

As in Section 0.3.1 it can be shown that the optimal makespan $\textsc{Opt}^\#$ of $I^\#$ fulfills $\textsc{Opt}^\# \le (1+\varepsilon)\textsc{Opt}$. The main difference in the argument is that the big jobs in $I^\#$ may be slightly smaller than their counterparts in instance I. But this actually works in our favor, since replacing jobs by slightly smaller ones can only decrease the makespan (or leave it unchanged), but can never increase it. With (0.6) and the definition $\varepsilon = 1/E$, we get

$$\textsc{Opt}^\# \;\le\; 2(1+\varepsilon)L \;=\; (2E^2+2E)\,\varepsilon^2 L. \tag{0.7}$$

In $I^\#$ the processing time of a rounded big job lies between εL and L. Hence, it is of the form $k\varepsilon^2 L$ where k is an integer with $E \le k \le E^2$. Note that $\varepsilon L = E\varepsilon^2 L$, and thus also the length of the chunks can be written in the form $k\varepsilon^2 L$. For $k = E, \ldots, E^2$, we denote by n_k the number of jobs in $I^\#$ whose processing time equals $k\varepsilon^2 L$. Notice that $n \ge \sum_{k=E}^{E^2} n_k$. A compact way of representing $I^\#$ is by collecting all the data in the vector $\vec{n} = [n_E, \ldots, n_{E^2}]$.

(B) How to solve the simplified instance. This step is more demanding than the trivial solution we used in Section 0.3.1. We will formulate the simplified instance as an integer linear program whose number of integer variables is bounded by some constant in E (and thus is independent of the input size). And then we can apply machinery from the literature for integer linear programs of fixed dimension to get a solution in polynomial time!

We need to introduce some notation to describe the possible schedules for instance $I^{\#}$. The *packing pattern* of a machine is a vector $\vec{u} = [u_E, \ldots, u_{E^2}]$, where u_k is the number of jobs of length $k\varepsilon^2 L$ assigned to that machine. For every vector $\vec{u}$ we denote $C(\vec{u}) = \sum_{k=E}^{E^2} u_k \cdot k$. Note that the workload of the corresponding machine equals $\sum_{k=E}^{E^2} u_k \cdot k\varepsilon^2 L = C(\vec{u})\,\varepsilon^2 L$. We denote by U the set of all packing patterns $\vec{u}$ for which $C(\vec{u}) \leq 2E^2 + 2E$, i.e., for which the corresponding machine has a workload of at most $(2E^2 + 2E)\,\varepsilon^2 L$. Because of inequality (0.7), we only need to consider packing patterns in U if we want to solve $I^{\#}$ to optimality. Since each job has length at least $E\varepsilon^2 L$, each packing pattern $\vec{u} \in U$ consists of at most $2E+2$ jobs. Therefore $|U| \leq (E^2 - E + 1)^{2E+3}$ holds, and the cardinality of U is bounded by a constant in E $(= 1/\varepsilon)$ that does not depend on the input. This property is important, as our integer linear program will have $2|U| + 1$ integer variables.

Now consider some fixed schedule for instance $I^{\#}$. For each vector $\vec{u} \in U$, we denote by $x_{\vec{u}}$ the numbers of machines with packing pattern $\vec{u}$. The 0-1-variable $y_{\vec{u}}$ serves as an indicator variable for $x_{\vec{u}}$: It takes the value 0 if $x_{\vec{u}} = 0$, and it takes the value 1 if $x_{\vec{u}} \geq 1$. Finally, we use the variable z to denote the makespan of the schedule. The integer linear program (ILP) is depicted in Figure 0.4.

$$
\begin{array}{lll}
\min & z & \\
\text{s.t.} & \sum_{\vec{u} \in U} x_{\vec{u}} \;=\; m & \\
& \sum_{\vec{u} \in U} x_{\vec{u}} \cdot \vec{u} \;=\; \vec{n} & \\
& y_{\vec{u}} \;\leq\; x_{\vec{u}} \;\leq\; m \cdot y_{\vec{u}} & \forall\, \vec{u} \in U \\
& C(\vec{u}) \cdot y_{\vec{u}} \;\leq\; z & \forall\, \vec{u} \in U \\
& x_{\vec{u}} \geq 0,\; x_{\vec{u}} \text{ integer} & \forall\, \vec{u} \in U \\
& y_{\vec{u}} \in \{0, 1\} & \forall\, \vec{u} \in U \\
& z \geq 0,\; z \text{ integer} &
\end{array}
$$

Figure 0.4: The integer linear program (ILP) in Section 0.3.2.

The objective is to minimize the value z; the makespan of the underlying schedule will be $z\varepsilon^2 L$ which is proportional to z. The first constraint states that exactly m machines must be used. The second constraint (which in fact is a set of $E^2 - E + 1$ constraints) ensures that all jobs can be packed; recall that $\vec{n} = [n_E, \ldots, n_{E^2}]$. The third set of constraints ties $x_{\vec{u}}$ to its indicator variable $y_{\vec{u}}$: $x_{\vec{u}} = 0$ implies $y_{\vec{u}} = 0$, and $x_{\vec{u}} \geq 1$ implies $y_{\vec{u}} = 1$ (note that each variable $x_{\vec{u}}$ takes an integer value between 0 and m). The fourth set of constraints ensures that $z\varepsilon^2 L$ is at least as large as the makespan of the underlying schedule: If the packing pattern $\vec{u}$ is not used in the schedule, then $y_{\vec{u}} = 0$ and the constraint boils down to $z \geq 0$. If the packing pattern $\vec{u}$ is used in the schedule, then $y_{\vec{u}} = 1$ and the constraint becomes $z \geq C(\vec{u})$ which is equivalent to $z\varepsilon^2 L \geq C(\vec{u})\,\varepsilon^2 L$. The remaining constraints are just integrality and non-negativity constraints. Clearly, the optimal makespan $\mathrm{OPT}^{\#}$ of $I^{\#}$ equals $z^*\varepsilon^2 L$ where z^* is the optimal objective value z^* of (ILP).

The number of integer variables in (ILP) is $2|U| + 1$, and we have already observed that this is a constant that does not depend at all on the instance I. We now apply Lenstra's famous algorithm [57] to solve (ILP). The time complexity of Lenstra's algorithm is exponential in the the number of variables, but polynomial in the logarithms of the coefficients. The coefficients in (ILP) are at most $\max\{m, n, 2E^2 + 2E\}$, and so (ILP) can be solved within an overall time complexity of $O(\log^{O(1)}(m + n))$. Note that here the hidden constant depends exponentially on $1/\varepsilon$. To summarize, we can solve the simplified instance $I^{\#}$ in polynomial time.

(C) How to translate the solution back. We proceed as in step (C) in Section 0.3.1: If in an optimal schedule for $I^{\#}$ the total size of chunks on machine M_i equals $S_i^{\#}$, then we reserve a time interval of length $S_i^{\#} + 2\varepsilon L$ on M_i and greedily pack the small jobs into all these reserved intervals. There is sufficient space to pack all small jobs. Big jobs in $I^{\#}$ are replaced by their counterparts in I. Since $p_j \leq (1 + \varepsilon)p_j^{\#}$ holds, this may increase the total processing time of big jobs on M_i by at most a factor of $1 + \varepsilon$. Similarly as in (0.5) we conclude that

$$\begin{aligned} L_i &\leq (1+\varepsilon)B_i^{\#} + S_i^{\#} + 2\varepsilon L \;\leq\; (1+\varepsilon)\mathrm{OPT}_i^{\#} + 2\varepsilon L \\ &\leq (1+\varepsilon)^2\mathrm{OPT} + 2\varepsilon\mathrm{OPT} \;=\; (1 + 4\varepsilon + \varepsilon^2)\mathrm{OPT}. \end{aligned}$$

Here we used $\mathrm{OPT}^{\#} \leq (1 + \varepsilon)\mathrm{OPT}$, and we used (0.6) to bound L from above. Since the term $4\varepsilon + \varepsilon^2$ can be made arbitrarily close to 0, this yields the PTAS.

Discussion. The polynomial time complexity of the above PTAS heavily relies on solving (ILP) by Lenstra's method, which really is heavy machinery. Are there other, simpler possibilities for solving the simplified instance $I^{\#}$ in polynomial time? Yes, there are. Hochbaum & Shmoys [37] use a dynamic programming approach with time complexity polynomial in n. However, the degree of this polynomial time complexity is proportional to $|U|$. This dynamic program is outlined in Exercise 0.3.3 in Section 0.3.4.

0.3.3 Total tardiness on a single machine

The problem. In the scheduling problem $1\,|\,|\sum T_j$, the input consists of n jobs J_j ($j = 1, \ldots, n$) with positive integer processing times p_j and integer due dates d_j. All jobs are available for processing at time zero, and preemption is not allowed. We denote by C_j the completion time of job J_j in some fixed schedule. Then the *tardiness* of job J_j in this schedule is $T_j = \max\{0, C_j - d_j\}$, i.e., the amount of time by which J_j violates its deadline. The goal is to schedule the jobs on a single machine such that the total tardiness $\sum_{j=1}^{n} T_j$ is minimized. Clearly, in an optimal schedule the jobs will be processed without any idle time between consecutive jobs and the optimal schedule can be fully specified by a permutation π^* of the n jobs. We use OPT to denote the objective value of the optimal schedule π^*.

The problem $1 \,||\, \sum T_j$ is NP-hard in the ordinary sense (Du & Leung [18]). Lawler [53] developed a dynamic programming formulation for $1 \,||\, \sum T_j$. The (pseudo-polynomial) time complexity of this dynamic program is $O(n^5 T_{\mathrm{EDD}})$. Here T_{EDD} denotes the maximum tardiness in the EDD-schedule, i.e., the schedule produced by the earliest-due-date rule (EDD-rule). The EDD-rule sequences the jobs in order of non-decreasing due date; this rule is easy to implement in polynomial time $O(n \log n)$, and it is well-known that the resulting EDD-schedule minimizes the maximum tardiness $\max T_j$. Hence T_{EDD} can be computed in polynomial time; this is important, as our simplification step will explicitly use the value of T_{EDD}. In case $T_{\mathrm{EDD}} = 0$ holds, all jobs in the EDD-schedule have tardiness 0 and the EDD-schedule constitutes an optimal solution to problem $1 \,||\, \sum T_j$. Therefore, we will only deal with the case $T_{\mathrm{EDD}} > 0$. Moreover, since in the schedule that minimizes the total tardiness, the most tardy job has tardiness at least T_{EDD}, we have

$$T_{\mathrm{EDD}} \;\leq\; \text{OPT}. \tag{0.8}$$

We will not discuss any details of Lawler's dynamic program here, since we will only use it as a black box. For our purposes, it is sufficient to know its time complexity $O(n^5 T_{\mathrm{EDD}})$ in terms of n and T_{EDD}.

(A) How to simplify an instance. Following Lawler [54] we will now add structure to the input, and thus eventually get an FPTAS. The additional structure depends on the following parameter Z.

$$Z \;:=\; \frac{2\varepsilon}{n(n+3)} \cdot T_{\mathrm{EDD}}$$

Note that $T_{\mathrm{EDD}} > 0$ yields $Z > 0$. We translate an arbitrary instance I of $1 \,||\, \sum T_j$ into a corresponding simplified instance $I^{\#}$. The processing time of the j-th job in $I^{\#}$ equals $p_j^{\#} = \lfloor p_j / Z \rfloor$, and its due date equals $d_j^{\#} = \lceil d_j / Z \rceil$.

The alert reader will have noticed that scaling the data by Z causes the processing times and due dates in instance $I^{\#}$ to be very far away from the corresponding processing times and due dates in the original instance I. How can we claim that $I^{\#}$ is a simplified version of I when it is so far away from instance I? One way of looking at this situation is that in fact we produce the simplified instance $I^{\#}$ in two steps. In the first step, the processing times in I are rounded down to the next integer multiple of Z, and the due dates are rounded up to the next integer multiple of Z. This yields the intermediate instance I' in which all the data are divisible by Z. In the second step we scale all processing times and due dates in I' by Z, and thus arrive at the instance $I^{\#}$ with much smaller numbers. Up to the scaling by Z, the instances I' and

$I^{\#}$ are equivalent. See Figure 0.5 for a listing of the variables in these three instances.

Instance	I	I'	$I^{\#}$
length of J_j	p_j	$p'_j = \lfloor p_j/Z \rfloor Z$ $p'_j \le p_j$	$p^{\#}_j = \lfloor p_j/Z \rfloor$ $p^{\#}_j = p'_j/Z$
due date of J_j	d_j	$d'_j = \lceil d_j/Z \rceil Z$ $d'_j \ge d_j$	$d^{\#}_j = \lceil d_j/Z \rceil$ $d^{\#}_j = d'_j/Z$
optimal value	$\textsc{Opt}$	$\textsc{Opt}'$ $(\le \textsc{Opt})$	$\textsc{Opt}^{\#}$ $(= \textsc{Opt}'/Z)$

Figure 0.5: The notation used in the PTAS for $1||\sum T_j$ in Section 0.3.3.

(B) How to solve the simplified instance. We solve instance $I^{\#}$ by applying Lawler's dynamic programming algorithm, whose time complexity depends on the number of jobs and on the maximum tardiness in the EDD-schedule. Clearly, there are n jobs in instance $I^{\#}$. What about the maximum tardiness? Let us consider the EDD-sequence π for the original instance I. When we move to I' and to $I^{\#}$, all the due dates are changed in a monotone way and therefore π is also an EDD-sequence for the jobs in the instances I' and $I^{\#}$. When we move from I to I', processing times cannot increase and due dates cannot decrease. Therefore, $T'_{\mathrm{EDD}} \le T_{\mathrm{EDD}}$. Since $I^{\#}$ results from I' by simple scaling, we conclude that

$$T^{\#}_{\mathrm{EDD}} \;=\; T'_{\mathrm{EDD}}/Z \;\le\; T_{\mathrm{EDD}}/Z \;=\; n(n+3)/(2\varepsilon).$$

Consequently $T^{\#}_{\mathrm{EDD}}$ is $O(n^2/\varepsilon)$. With this the time complexity of Lawler's dynamic program for $I^{\#}$ becomes $O(n^7/\varepsilon)$, which is polynomial in n and in $1/\varepsilon$. And that is exactly the type of time complexity that we need for an FPTAS! So, the simplified instance is indeed easy to solve.

(C) How to translate the solution back. Consider an optimal job sequence for instance $I^{\#}$. To simplify the presentation, we renumber the jobs in such a way that this optimal sequence becomes $J_1, J_2, J_3, \ldots, J_n$. Translating this optimal solution back to the original instance I is easy: We take the same sequence for the jobs in I to get an approximate solution. We now want to argue that the objective value of this approximate solution is very close to the optimal objective value. This is done by exploiting the structural similarities between the three instances I, I', and $I^{\#}$.

We denote by C_j and T_j the completion time and the tardiness of the j-th job in the schedule for I, and by $C^{\#}_j$ and $T^{\#}_j$ the completion time and the tardiness of the j-th job in the schedule for instance $I^{\#}$. By the definition of

$p_j^\#$ and $d_j^\#$, we have $p_j \le Zp_j^\# + Z$ and $d_j > Zd_j^\# - Z$. This yields for the completion times C_j ($j = 1, \ldots, n$) in the approximate solution that

$$C_j \;=\; \sum_{i=1}^{j} p_j \;\le\; Z\sum_{i=1}^{j} p_j^\# + jZ \;=\; Z \cdot C_j^\# + jZ.$$

As a consequence,

$$T_j \;=\; \max\{0,\, C_j - d_j\} \;\le\; \max\{0,\, (Z\cdot C_j^\# + jZ) - (Zd_j^\# - z)\} \;\le\; Z\cdot T_j^\# + (j+1)Z.$$

This finally leads to

$$\sum_{j=1}^{n} T_j \;\le\; \sum_{j=1}^{n} Z \cdot T_j^\# + \frac{1}{2}n(n+3)\cdot Z \;\le\; \textsc{Opt} + \varepsilon T_{\mathrm{EDD}} \;\le\; (1+\varepsilon)\textsc{Opt}.$$

Let us justify the correctness of the last two inequalities above. The penultimate inequality is justified by comparing instance I' to instance I. Since I' is a scaled version of $I^\#$, we have $\sum_{j=1}^{n} Z \cdot T_j^\# = Z \cdot \textsc{Opt}^\# = \textsc{Opt}'$. Since in I' the jobs are shorter and have less restrictive due dates than in I, we have $\textsc{Opt}' \le \textsc{Opt}$. The last inequality follows from the observation (0.8) in the beginning of this section. To summarize, for each $\varepsilon > 0$ we can find within a time of $O(n^7/\varepsilon)$ an approximate schedule whose objective value is at most $(1+\varepsilon)\textsc{Opt}$. And that is exactly what is needed for an FPTAS!

0.3.4 Exercises

Exercise 0.3.1. Construct a PTAS for $Pm\,|\,|\,C_{\max}$ by appropriately modifying the approach described in Section 0.3.1. Work with $L = \max\{\frac{1}{m}p_{\mathrm{sum}}, p_{\max}\}$, and use the analogous definition of big and small jobs in the simplification step (A). Argue that the inequality (0.3) is again satisfied. Modify the translation step (C) appropriately so that all small jobs are packed. What is your worst case guarantee in terms of ε and m? How does your time complexity depend on ε and m?

Exercise 0.3.2. In the PTAS for $P2\,|\,|\,C_{\max}$ in Section 0.3.1, we replaced the small jobs in instance I by lots of chunks of length εL in instance $I^\#$. Consider the following alternative way of handling the small jobs in I: Put all the small jobs into a canvas bag. While there are at least two jobs with lengths smaller than εL in the bag, merge two such jobs. That is, repeatedly replace two jobs with processing times $p', p'' \le \varepsilon L$ by a single new job of length $p' + p''$. The simplified instance $I_{\mathrm{alt}}^\#$ consists of the final contents of the bag.

Will this lead to another PTAS for $P2\,|\,|\,C_{\max}$? Does the inequality (0.3) still hold true? How can you bound the number of jobs in the simplified instance $I_{\mathrm{alt}}^\#$? How would you translate an optimal schedule for $I_{\mathrm{alt}}^\#$ into an approximate schedule for I?

Exercise 0.3.3. This exercise deals with the dynamic programming approach of Hochbaum & Shmoys [37] for the simplified instance $I^{\#}$ of $P\,|\,|\,C_{\max}$. This dynamic progam has already been mentioned at the end of Section 0.3.2, and we will use the notation of this section in this exercise.

Let V be the set of integer vectors that encode subsets of the jobs in $I^{\#}$, i.e. $V = \{\vec{v} : \vec{0} \le \vec{v} \le \vec{n}\}$. For every $\vec{v} \in V$ and for every i, $0 \le i \le m$, denote by $F(i, \vec{v})$ the makespan of the optimal schedule for the jobs in $\vec{v}$ on exactly i machines that only uses packing patterns from U. If no such schedule exists we set $F(i, \vec{v}) = +\infty$. For example, $F(1, \vec{v}) = C(\vec{v})\,\varepsilon^2 L$ if $\vec{v} \in U$, and $F(1, \vec{v}) = +\infty$ if $\vec{v} \notin U$.

(a) Show that the cardinality of V is bounded by a polynomial in n. How does the degree of this polynomial depend on ε and E?

(b) Find a recurrence relation that for $i \ge 2$ and $\vec{v} \in V$ expresses the value $F(i, \vec{v})$ in terms of the values $F(i-1, \vec{v} - \vec{u})$ with $\vec{u} \le \vec{v}$.

(c) Use this recurrence relation to compute the value $F(m, \vec{n})$ in polynomial time. How does the time complexity depend on n?

(d) Argue that the optimal makespan for $I^{\#}$ equals $F(m, \vec{n})$. How can you get the corresponding optimal schedule? [Hint: Store some extra information in the dynamic program.]

Exercise 0.3.4. Consider n jobs J_j ($j = 1, \ldots, n$) with positive integer processing times p_j on three identical machines. The goal is to find a schedule with machine loads L_1, L_2, L_3 that minimizes the value $L_1^2 + L_2^2 + L_3^2$, the sum of squared machine loads.

Construct a PTAS for this problem by following the approach described in Section 0.3.1. Construct a PTAS for minimizing the sum of squared machine loads when the number m of machines is part of the input by following the approach described in Section 0.3.2. For a more general discussion of this problem, see Alon, Azar, Woeginger & Yadid [3].

Exercise 0.3.5. Consider n jobs J_j ($j = 1, \ldots, n$) with positive integer processing times p_j on three identical machines M_1, M_2, and M_3, together with a positive integer T. You have already agreed to lease all three machines for T time units. Hence, if machine M_i completes at time $L_i \le T$, then your cost for this machine still is proportional to T. If machine M_i completes at time $L_i > T$, then you have to pay extra for the overtime, and your cost is proportional to L_i. To summarize, your goal is to minimize the value $\max\{T, L_1\} + \max\{T, L_2\} + \max\{T, L_3\}$, your overall cost.

Construct a PTAS for this problem by following the approach described in Section 0.3.1. Construct a PTAS for minimizing the corresponding objective function when the number m of machines is part of the input by following the approach described in Section 0.3.2. For a more general discussion of this problem, see Alon, Azar, Woeginger & Yadid [3].

Exercise 0.3.6. Consider n jobs J_j $(j = 1, \ldots, n)$ with positive integer processing times p_j on two identical machines. The goal is to find a schedule with machine loads L_1 and L_2 that minimizes the following objective value:

(a) $|L_1 - L_2|$

(b) $\max\{L_1, L_2\}/\min\{L_1, L_2\}$

(c) $(L_1 - L_2)^2$

(d) $L_1 + L_2 + L_1 \cdot L_2$

(e) $\max\{L_1, L_2/2\}$

For which of these problems can you get a PTAS? Can you prove statements analogous to inequality (0.3) in Section 0.3.1? For the problems where you fail to get a PTAS, discuss where and why the approach of Section 0.3.1 breaks down.

Exercise 0.3.7. Consider two instances I and I' of $1\,|\,|\sum T_j$ with n jobs, processing times p_j and p'_j, and due dates d_j and d'_j. Denote by OPT and OPT' the respective optimal objective values. Furthermore, let $\varepsilon > 0$ be a real number. Prove or disprove:

(a) If $p_j \le (1+\varepsilon)p'_j$ and $d_j = d'_j$ for $1 \le j \le n$, then $\mathrm{OPT} \le (1+\varepsilon)\mathrm{OPT}'$.

(b) If $d_j \ge (1+\varepsilon)d'_j$ and $p_j \le (1+\varepsilon)p'_j$ for $1 \le j \le n$, then $\mathrm{OPT} \le (1+\varepsilon)\mathrm{OPT}'$.

(c) If $d'_j \le (1-\varepsilon)d_j$ and $p_j = p'_j$ for $1 \le j \le n$, then $\mathrm{OPT} \le (1-\varepsilon)\mathrm{OPT}'$.

Exercise 0.3.8. In the problem $1\,|\,|\sum w_jT_j$ the input consists of n jobs J_j $(j = 1, \ldots, n)$ with positive integer processing times p_j, integer due dates d_j, and positive integer weights w_j. The goal is to schedule the jobs on a single machine such that the total weighted tardiness $\sum w_jT_j$ is minimized.

Can you modify the approach in Section 0.3.3 so that it yields a PTAS for $1\,|\,|\sum w_jT_j$? What are the main obstacles? Consult the paper by Lawler [53] to learn more about the computational complexity of $1\,|\,|\sum w_jT_j$!

Exercise 0.3.9. Consider the problem $1\,|\,r_j\,|\sum C_j$ whose input consists of n jobs J_j $(j = 1, \ldots, n)$ with processing times p_j and release dates r_j. In a feasible schedule for this problem, no job J_j is started before its release date r_j. The goal is to find a feasible schedule of the jobs on a single machine such that the total job completion time $\sum C_j$ is minimized.

Consider two instances I and I' of $1\,|\,r_j\,|\sum C_j$ with n jobs, processing times p_j and p'_j, and release dates r_j and r'_j. Denote by OPT and OPT' the respective optimal objective values. Furthermore, let $\varepsilon > 0$ be a real number. Prove or disprove:

(a) If $p_j \le (1+\varepsilon)p'_j$ and $r_j = r'_j$ for $1 \le j \le n$, then $\mathrm{OPT} \le (1+\varepsilon)\mathrm{OPT}'$.

(b) If $r_j \le (1+\varepsilon)r'_j$ and $p_j = p'_j$ for $1 \le j \le n$, then $\text{OPT} \le (1+\varepsilon)\text{OPT}'$.

(c) If $r_j \le (1+\varepsilon)r'_j$ and $p_j \le (1+\varepsilon)p'_j$ for $1 \le j \le n$, then $\text{OPT} \le (1+\varepsilon)\text{OPT}'$.

Exercise 0.3.10. An instance of the flow shop problem $F3\,|\,|\,C_{\max}$ consists of three machines M_1, M_2, M_3 together with n jobs $J_1, \dots, J_n$. Every job J_j first has to be processed for $p_{j,1}$ time units on machine M_1, then (an arbitrary time later) for $p_{j,2}$ time units on M_2, and finally (again an arbitrary time later) for $p_{j,3}$ time units on M_3. The goal is to find a schedule that minimizes the makespan. In the closely related no-wait flow shop problem $F3\,|\,\textit{no-wait}\,|\,C_{\max}$, there is no waiting time allowed between the processing of a job on consecutive machines.

Consider two flow shop instances I and I' with processing times $p_{j,i}$ and $p'_{j,i}$ such that $p_{j,i} \le p'_{j,i}$ holds for $1 \le j \le n$ and $1 \le i \le 3$.

(a) Prove: In the problem $F3\,|\,|\,C_{\max}$, the optimal objective value of I is always less or equal to the optimal objective value of I'.

(b) Disprove: In the problem $F3\,|\,\textit{no-wait}\,|\,C_{\max}$, the optimal objective value of I is always less or equal to the optimal objective value of I'. [Hint: Look for a counter-example with three jobs.]

0.4 Structuring the output

As second standard approach to the construction of approximation schemes we discuss the technique of *adding structure to the output*. Here the main idea is to cut the output space (i.e., the set of feasible solutions) into lots of smaller regions over which the optimization problem is easy to approximate. Tackling the problem separately for each smaller region and taking the best approximate solution over all regions will then yield a globally good approximate solution. More formally, the approach can be described by the following three-step procedure; see Figure 0.6 for an illustration.

(A) Partition. Partition the feasible solution space $\mathcal{F}$ of instance I into a number of *districts* $\mathcal{F}^{(1)}, \mathcal{F}^{(2)}, \dots, \mathcal{F}^{(d)}$ such that $\bigcup_{\ell=1}^{d} \mathcal{F}^{(\ell)} = \mathcal{F}$. This partition depends on the desired precision ε of approximation. The closer ε is to zero, the finer should this partition be. The number d of districts must be polynomially bounded in the size of the input.

(B) Find representatives. For each district $\mathcal{F}^{(\ell)}$ determine a *good representative* whose objective value $\text{APP}^{(\ell)}$ is a good approximation of the optimal objective value $\text{OPT}^{(\ell)}$ in $\mathcal{F}^{(\ell)}$. The time needed for finding the representative must be polynomial in the input size.

(C) Take the best. Select the best of all representatives as approximate solution with objective value APP for instance I.

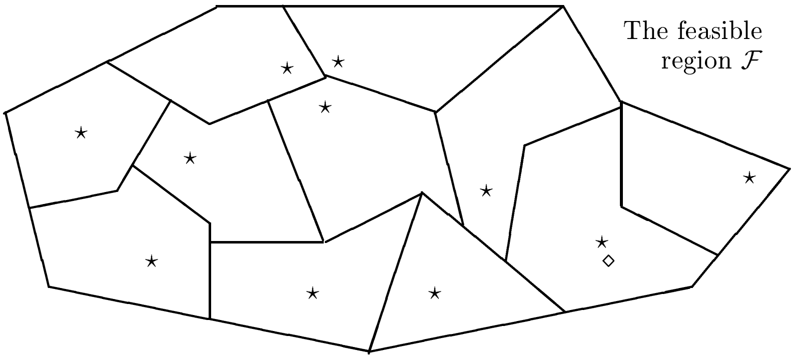

Figure 0.6: Structuring the output. It is difficult to optimize over the entire feasible region $\mathcal{F}$. Hence, one cuts $\mathcal{F}$ into many districts and finds a good representative (symbolized by $\star$) for each district. Locally, i.e., within their districts, the representatives are excellent approximate solutions. The best of the representatives constitutes a good approximation of the globally optimal solution (symbolized by $\diamond$).

The overall time complexity of this approach is polynomial: There is a polynomial number of districts, and each district is handled in polynomial time in step (B). Step (C) optimizes over a polynomial number of representatives. The globally optimal solution with objective value $\textsc{Opt}$ must be contained in at least one of the districts, say in district $\mathcal{F}^{(\ell)}$. Then $\textsc{Opt} = \textsc{Opt}^{(\ell)}$. Since the representative for $\mathcal{F}^{(\ell)}$ gives a good approximation of $\textsc{Opt}^{(\ell)}$, it also yields a good approximation of the global optimum. Hence, also the final output of the algorithm will be a good approximation of the global optimum. Note that this argument still works, if we only compute representatives for districts that contain a gobal optimum. Although generally it is difficult to identify the districts that contain a gobal optimum, there sometimes is some simple a priori reason why a certain district cannot contain a global optimum. In this case we can save time by excluding the district from further computations (see Section 0.4.2 for an illustration of this idea).

The whole approach hinges on the choice of the districts in step (A). If the partition chosen is too fine (for instance, if every feasible solution in $\mathcal{F}$ forms its own district), then we will have an exponential number of districts and steps (B) and (C) probably cannot be done in polynomial time. If the partition chosen is too crude (for instance, if the set $\mathcal{F}$ itself forms the only district), then the computation of the representatives in step (B) will be about as difficult as finding an approximation scheme for the original problem. The idea is to work with a moderately small number of districts. Here moderately small means polynomial in the size of the instance I, exactly as required in the statement of step (A). In every district, all the feasible solutions share a certain common property (for instance, some district may contain all feasible solutions that run the same job J_{11} at time 8 on machine M_3). This common property fixes several parameters of the feasible solutions in the district, whereas other parameters are still free. And in the ideal case, it is easy to approximately optimize the remaining free parameters.

To our knowledge the approach of structuring the output was first used in 1969 in a paper by Graham [27] on makespan minimization on identical machines. Remarkably, in the last section of his paper Graham attributes the approach to a suggestion by Dan Kleitman and Donald E. Knuth. So it seems that this approach has many fathers. Hochbaum & Maass [36] use the approach of structuring the output to get a PTAS for covering a Euclidean point set by the minimum number of unit-squares. In the 1990s, Leslie Hall and David Shmoys wrote a sequence of very influential papers (see Hall [29, 31], Hall & Shmoys [32, 33, 34]) that all are based on the concept of a so-called *outline*. An outline is a compact way of specifying the common properties of a set of schedules that form a district. Hence, approximation schemes based on outlines follow the approach of structuring the output.

In the following two sections, we will illustrate the technique of simplifying the output with the help of two examples. Section 0.4.1 deals with makespan minimization on two identical machines; we will discuss the arguments of Graham [27]. Section 0.4.2 deals with makespan minimization on two unrelated machines. Section 0.4.3 lists several exercises.

0.4.1 Makespan on two identical machines

The problem. We return to the problem $P2\,|\,|\,C_{\max}$ that was introduced and thoroughly discussed in Section 0.3.1: There are n jobs J_j ($j = 1, \ldots, n$) with processing times p_j, and the goal is to find a schedule on two identical machines that minimizes the makespan. Again, we denote $p_{\text{sum}} = \sum_{j=1}^{n} p_j$, $p_{\max} = \max_{j=1}^{n} p_j$, and $L = \max\{\frac{1}{2}p_{\text{sum}}, p_{\max}\}$ with

$$L \;\le\; \text{OPT}. \tag{0.9}$$

In this section we will construct another PTAS for $P2\,|\,|\,C_{\max}$, but this time the PTAS will be based on the technique of structuring the output. We will roughly follow the argument in the paper of Graham [27] from 1969.

(A) How to define the districts. Let I be an instance of $P2\,|\,|\,C_{\max}$, and let $\varepsilon > 0$ be a precision parameter. Recall from Section 0.3.1 that a *small job* is a job with processing time at most εL, that a *big job* is a job with processing time strictly greater than εL, and that there are at most $2/\varepsilon$ big jobs in I. Consider the set $\mathcal{F}$ of feasible solutions for I. Every feasible solution $\sigma \in \mathcal{F}$ specifies an assignment of the n jobs to the two machines.

We define the districts $\mathcal{F}^{(1)}, \mathcal{F}^{(2)}, \ldots$ according to the assignment of the big jobs to the two machines: Two feasible solutions σ_1 and σ_2 lie in the same district if and only if σ_1 assigns every big jobs to the same machine as σ_2 does. Note that the assignment of the small jobs remains absolutely free. Since there are at most $2/\varepsilon$ big jobs, there are at most $2^{2/\varepsilon}$ different ways for assigning these jobs to two machines. Hence, the number of districts in our partition is bounded by a fixed constant whose value is independent of the input size. Perfect! In order to make the approach work, we could have afforded that the number of districts grows polynomially with the size of I, but we even manage to get along with only a constant number of districts!

(B) How to find good representatives. Consider a fixed district $\mathcal{F}^{(\ell)}$, and denote by $\mathrm{OPT}^{(\ell)}$ the makespan of the best schedule in this district. In $\mathcal{F}^{(\ell)}$ the assignments of the big jobs to their machines are fixed, and we denote by $B_i^{(\ell)}$ $(i = 1, 2)$ the total processing time of big jobs assigned to machine M_i. Clearly,

$$T \;:=\; \max\{B_1^{(\ell)}, B_2^{(\ell)}\} \;\leq\; \mathrm{OPT}^{(\ell)}. \tag{0.10}$$

Our goal is to determine in polynomial time some schedule in $\mathcal{F}^{(\ell)}$ with makespan $\mathrm{APP}^{(\ell)} \leq (1+\varepsilon)\mathrm{OPT}^{(\ell)}$. Only the small items remain to be assigned. Small items behave like sand, and it is easy to pack this sand in a very dense way by sprinkling it across the machines. More formally, we do the following: The initial workload of machines M_1 and M_2 are $B_1^{(\ell)}$ and $B_2^{(\ell)}$, respectively. We assign the small jobs one by one to the machines; every time a job is assigned, it is put on the machine with the currently smaller workload (ties are broken arbitrarily). The resulting schedule $\sigma^{(\ell)}$ with makespan $\mathrm{APP}^{(\ell)}$ is our representative for the district $\mathcal{F}^{(\ell)}$. Clearly, $\sigma^{(\ell)}$ is computable in polynomial time.

How close is $\mathrm{APP}^{(\ell)}$ to $\mathrm{OPT}^{(\ell)}$? In case $\mathrm{APP}^{(\ell)} = T$ holds, the inequality (0.10) yields that $\sigma^{(\ell)}$ in fact is an optimal schedule for the district $\mathcal{F}^{(\ell)}$. In case $\mathrm{APP}^{(\ell)} > T$ holds, we consider the machine M_i with higher workload in the schedule $\sigma^{(\ell)}$. Then the last job that was assigned to M is a small job and thus has processing time at most εL. At the moment when this small job was assigned to M_i, the workload of M_i was at most $\frac{1}{2}p_{\mathrm{sum}}$. By using (0.9) we get that

$$\mathrm{APP}^{(\ell)} \;\leq\; \frac{1}{2}p_{\mathrm{sum}} + \varepsilon L \;\leq\; (1+\varepsilon)L \;\leq\; (1+\varepsilon)\mathrm{OPT} \;\leq\; (1+\varepsilon)\mathrm{OPT}^{(\ell)}.$$

Hence, in either case the makespan of the representative is at most $1+\varepsilon$ times the optimal makespan in $\mathcal{F}^{(\ell)}$. Since the selection step (C) is trivial to do, we get the PTAS.

Discussion. Let us compare our new PTAS for $P2\,|\,|\,C_{\max}$ to the old PTAS for $P2\,|\,|\,C_{\max}$ from Section 0.3.1 that was based on the technique of structuring the input. An obvious similarity is that both approximation schemes classify the jobs into big ones and small ones, and then treat big jobs differently from small jobs. Another similarity is that the time complexity of both approximation schemes is linear in n, but exponential in $1/\varepsilon$. But apart from this, the two approaches are very different: The old PTAS manipulates and modifies the instance I until it becomes trivial to solve, whereas the strategy of the new PTAS is to distinguish lots of cases that all are relatively easy to handle.

0.4.2 Makespan on two unrelated machines

The problem. In the scheduling problem $R2\,||\,C_{\max}$ the input consists of two unrelated machines A and B, together with n jobs J_j ($j = 1, \ldots, n$). If job J_j is assigned to machine A then its processing time is a_j, and if it is assigned to machine B then its processing time is b_j. The objective is to find a schedule that minimizes the makespan. The problem $R2\,||\,C_{\max}$ is NP-hard in the ordinary sense. We will construct a PTAS for $R2\,||\,C_{\max}$ that uses the technique of structuring the output. This section is based on Potts [67].

Let I be an instance of $R2\,||\,C_{\max}$, and let $\varepsilon > 0$ be a precision parameter. We denote $K = \sum_{j=1}^{n} \min\{a_j, b_j\}$, and we observe that

$$\frac{1}{2}K \;\leq\; \textsc{Opt} \;\leq\; K. \tag{0.11}$$

To see the lower bound in (0.11), we observe that in the optimal schedule job J_j will run for at least $\min\{a_j, b_j\}$ time units. Hence, the total processing time in the optimal schedule is at least K and the makespan is at least $\frac{1}{2}K$. To see the upper bound in (0.11), consider the schedule that assigns J_j to machine A if $a_j \leq b_j$ and to machine B if $a_j > b_j$. This schedule is feasible, and its makespan is at most K.

(A) How to define the districts. Consider the set $\mathcal{F}$ of feasible solutions for I. Every feasible solution $\sigma \in \mathcal{F}$ specifies an assignment of the n jobs to the two machines. A scheduled job in some feasible solution is called *big* if and only if its processing time is greater than εK. Note that this time we define big jobs only relative to a feasible solution! There is no obvious way of doing an absolute job classification that is independent of the schedules, since there might be jobs with $a_j > \varepsilon K$ and $b_j \leq \varepsilon K$, or jobs with $a_j \leq \varepsilon K$ and $b_j > \varepsilon K$. Would one call such a job big or small? Our definition is a simple way of avoiding this difficulty.

The districts are defined according to the assignment of the big jobs to the two machines: Two feasible solutions σ_1 and σ_2 lie in the same district if and only if σ_1 and σ_2 process the same big jobs on machine A and the same big jobs on machine B. For a district $\mathcal{F}^{(\ell)}$, we denote by $A^{(\ell)}$ the total processing time of big jobs assigned to machine A, and by $B^{(\ell)}$ the total processing time of big jobs assigned to machine B. We kill all districts $\mathcal{F}^{(\ell)}$ for which $A^{(\ell)} > K$ or $B^{(\ell)} > K$ holds, and we disregard them from further consideration. Because of inequality (0.11), these districts cannot contain an optimal schedule and hence are worthless for our investigation. Exercise 0.4.4 in Section 0.4.3 even demonstrates that killing these districts is essential, since otherwise the time complexity of the approach might explode and become exponential!

Now let us estimate the number of surviving districts. Since $A^{(\ell)} \leq K$ holds in every surviving district $\mathcal{F}^{(\ell)}$, at most $1/\varepsilon$ big jobs are assigned to machine A. By an analogous argument, at most $1/\varepsilon$ big jobs are assigned to machine B. Hence, the district is fully specified by up to $2/\varepsilon$ big jobs that are chosen from a pool of up to n jobs. This yields $O(n^{2/\varepsilon})$ surviving districts. Since ε is fixed and not part of the input, the number of districts is polynomially bounded in the size of the input.

(B) How to find good representatives. Consider some surviving district $\mathcal{F}^{(\ell)}$ in which the assignment of the big jobs has been fixed. The unassigned jobs belong to one of the following four types:

(i) $a_j \leq \varepsilon K$ and $b_j \leq \varepsilon K$,

(ii) $a_j > \varepsilon K$ and $b_j \leq \varepsilon K$,

(iii) $a_j \leq \varepsilon K$ and $b_j > \varepsilon K$,

(iv) $a_j > \varepsilon K$ and $b_j > \varepsilon K$.

If there is an unassigned job of type (iv), the district $\mathcal{F}^{(\ell)}$ is empty, and we may disregard it. If there are unassigned jobs of type (ii) or (iii), then we assign them in the obvious way without producing any additional big jobs. The resulting fixed total processing time on machines A and B is denoted by $\alpha^{(\ell)}$ and $\beta^{(\ell)}$, respectively. We renumber the jobs such that $J_1, \ldots, J_k$ with $k \leq n$ become the remaining unassigned jobs. Only jobs of type (i) remain to be assigned, and this is done by means of the integer linear program (ILP) and its relaxation (LPR) that both are depicted in Figure 0.7: For each job J_j with $1 \leq j \leq k$, there is a 0-1-variable x_j in (ILP) that encodes the assignment of J_k. If $x_j = 1$, then J_j is assigned to machine A, and if $x_j = 0$, then J_j is assigned to machine B. The variable z denotes the makespan of the schedule. The first and second constraints state that z is an upper bound on the total processing time on the machines A and B. The remaining constraints in (ILP) are just integrality constraints. The linear programming relaxation (LPR) is identical to (ILP), except that here the x_j are continuous variables in the interval $[0, 1]$.

	(ILP)		(LPR)
min	z	min	z
s.t.	$\alpha^{(\ell)} + \sum_{j=1}^{k} a_j x_j \leq z$	s.t.	$\alpha^{(\ell)} + \sum_{j=1}^{k} a_j x_j \leq z$
	$\beta^{(\ell)} + \sum_{j=1}^{k} b_j (1 - x_j) \leq z$		$\beta^{(\ell)} + \sum_{j=1}^{k} b_j (1 - x_j) \leq z$
	$x_j \in \{0, 1\} \qquad j = 1, \ldots, k.$		$0 \leq x_j \leq 1 \qquad j = 1, \ldots, k.$

Figure 0.7: The integer linear program (ILP) and its relaxation (LPR) in Section 0.4.2.

The integrality constraints on the x_j make it NP-hard to find a feasible solution for (ILP). But this does not really matter to us, since the linear programming relaxation (LPR) is easy to solve! We determine in polynomial time a basic feasible solution x_j^* with $j = 1, \ldots, n$ and z^* for (LPR). Assume that exactly f of the values x_j^* are fractional, and that the remaing $k - f$ values are integral. We want to analyze which of the $2k + 2$ inequalities of (LPR) may be fulfilled as an equality by the basic feasible solution: For every integral x_j^*, equality holds in exactly one of the two inequalities $0 \le x_j^*$ and $x_j^* \le 1$. For every fractional x_j^*, equality holds in none of the two inequalities $0 \le x_j^*$ and $x_j^* \le 1$. Moreover, the first two constraints may be fulfilled with equality. All in all, this yields that at most $k - f + 2$ of the constraints can be fulfilled with equality. On the other hand, a basic feasible solution is a vertex of the underlying polyhedron in $(k + 1)$-dimensional space. It is only determined if equality holds in at least $k + 1$ of the $2k + 2$ inequalities in (LPR). We conclude that $k + 1 \le k - f + 2$, which is equivalent to $f \le 1$.

We have shown that at most one of the values x_j^* is not integral. In other words, we have almost found a feasible solution for (ILP)! Now it is easy to get a good representative: Each job J_j with $x_j^* = 1$ is assigned to machine A, each J_j with $x_j^* = 0$ is assigned to machine B, and if there is a fractional x_j^* then the corresponding job is assigned to machine A. This increases the total processing time on A by at most εK. The makespan $\text{APP}^{(\ell)}$ of the resulting representative fulfills

$$\text{APP}^{(\ell)} \;\le\; z^* + \varepsilon K \;\le\; \text{OPT}^{(\ell)} + \varepsilon K.$$

Consider a district that contains an optimal solution with makespan OPT. Then the makespan of the corresponding representative is at most $\text{OPT} + \varepsilon K$ which by (0.11) is at most $(1 + 2\varepsilon)\text{OPT}$. To summarize, the selection step (C) will find a representative with makespan at most $(1 + 2\varepsilon)\text{OPT}$, and thus we get the PTAS.

0.4.3 Exercises

Exercise 0.4.1. Construct a PTAS for $Pm \,|\,|\, C_{\max}$ by appropriately modifying the approach described in Section 0.4.1. Classify the jobs according to $L = \max\{\frac{1}{m} p_{\text{sum}}, p_{\max}\}$, and define the districts according to the assignment of the big jobs. How many districts do you get? How do you compute good representatives in polynomial time? What is your worst case guarantee in terms of ε and m? How does your time complexity depend on ε and m?

Can you modify the PTAS from Section 0.4.1 so that it works for the problem $P \,|\,|\, C_{\max}$ with an arbitrary number of machines? What are the main obstacles in this approach?

Exercise 0.4.2. This exercise concerns the PTAS for $P2 \,|\,|\, C_{\max}$ in Section 0.4.1. Recall that in every district $\mathcal{F}^{(\ell)}$, all the schedules had the same workloads $B_1^{(\ell)}$ and $B_2^{(\ell)}$ of big jobs on the machines M_1 and M_2, respectively. We first computed a representative for every district by greedily adding the small jobs, and afterwards selected the globally best representative.

(a) Suppose that we mess up these two steps in the following way: First, we select the district $\mathcal{F}^{(\ell)}$ that minimizes $\max\{B_1^{(\ell)}, B_2^{(\ell)}\}$. We immediately abandon all the other districts. Then we compute the representative for the selected district by greedily adding the small jobs, and finally output this representative. Would this still yield a PTAS?

(b) Generalize your answer from (a) to the problem $Pm\,|\,|\,C_{\max}$. To do this, you probably should first work your way through Exercise 0.4.1.

Exercise 0.4.3. Consider n jobs J_j $(j = 1, \ldots, n)$ with positive integer processing times p_j on three identical machines. The goal is to find a schedule with machine loads L_1, L_2, and L_3 that minimizes the value $L_1^2 + L_2^2 + L_3^2$, i.e., the sum of squared machine loads.

(a) Construct a PTAS for this problem by following the approach described in Section 0.4.1.

(b) Now consider the messed up approach from Exercise 0.4.2 that first selects the 'best' district by only judging from the assignment of the big jobs, and afterwards outputs the representative for the selected district. Will this still yield a PTAS?

[Hint: For a fixed precision ε, define $E = \lceil 1/\varepsilon \rceil$. Investigate the instance $I(\varepsilon)$ that consists of six jobs with lengths $13, 9, 9, 6, 6, 6$ together with E tiny jobs of length $5/E$.]

(c) Does the messed up approach from Exercise 0.4.2 yield a PTAS for minimizing the sum of squared machine loads on $m = 2$ machines?

Exercise 0.4.4. When we defined the districts in Section 0.4.2, we decided to kill all districts $\mathcal{F}^{(\ell)}$ with $A^{(\ell)} > K$ or $B^{(\ell)} > K$. Consider an instance I of $R2\,|\,|\,C_{\max}$ with n jobs, where $a_j = 0$ and $b_j \equiv b > 0$ for $1 \le j \le n$. How many districts are there overall? How many of these districts are killed, and how many of them survive and get representatives?

Exercise 0.4.5. Construct a PTAS for $Rm\,|\,|\,C_{\max}$ via the approach described in Section 0.4.2. How do you define your districts? How many districts do you get? How many constraints are there in your integer linear program? What can you say about the number of fractional values in a basic feasible solution for the relaxation? How do you find the representatives?

Exercise 0.4.6. This exercise deals with problem $P3\,|\,pmtn\,|\,\mathrm{Rej}{+}C_{\max}$, a variant of parallel machine scheduling with job rejections. An instance consists of n jobs J_j $(j = 1, \ldots, n)$ with a rejection penalty e_j and a processing time p_j. The goal is to reject a subset of the jobs and to schedule the remaining jobs preemptively on three identical machines, such that the total penalty of all rejected jobs plus the makespan of the scheduled jobs becomes minimum. This problem is NP-hard in the ordinary sense (Hoogeveen, Skutella & Woeginger [41]). The optimal makespan for preemptive scheduling without rejections is the maximum of the length of the longest job and of the average machine load; see [55].

Design a PTAS for $P3\,|\,pmtn\,|\,\mathrm{Rej}+C_{\max}$! [Hint: First get upper and lower estimates on the optimal solution. Define big jobs with respect to these estimates. The districts are defined with respect to the rejected big jobs. As in Section 0.4.2, use a linear relaxation of an integer program to schedule the small jobs in the representatives.]

Exercise 0.4.7. This exercise asks you to construct a PTAS for the following variant of the knapsack problem. The input consists of n items with positive integer sizes a_j ($j = 1, \ldots, n$) and of a knapsack with capacity b. The goal is to pack the knapsack as full as possible but without overpacking it. In other words, the goal is to determine $J \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in J} a_j$ is maximized subject to the constraint $\sum_{j \in J} a_j \leq b$.

(a) Classify items into big ones (those with size greater than εb) and small ones (those with size at most εb). Define your districts via the set of big items that a feasible solution packs into the knapsack. What is the maximum number of big items that can fit into the knapsack? How many districts do you get?

(b) Explain how to find good representatives for the districts.

Exercise 0.4.8. This exercise asks you to construct a PTAS for the following problem. The input consists of n items with positive integer sizes a_j ($j = 1, \ldots, n$) and of two knapsacks with capacity b. The goal is to pack the maximum *number* of items into the two knapsacks.

(a) Define one special district that contains all feasible packings with up to $2/\varepsilon$ items. The remaining districts are defined according to the placement of the big items. How do you define the big items? And how many districts do you get?

(b) Explain how you find good representatives for the districts. How do you handle the special district?

Exercise 0.4.9. All the PTAS's we have seen so far were heavily based on some a priori upper and lower estimates on the optimal objective value. For instance, in Sections 0.3.1, 0.3.2, and Section 0.4.1 we used the bounds L and $2L$ on OPT, and in Section 0.4.2 we used the bounds $\frac{1}{2}K$ and K on OPT, and so on. This exercise discusses a general trick for getting a PTAS that works even if you only know very crude upper and lower bounds on the optimal objective value. The trick demonstrates that you may concentrate on the special situation where you have an a priori guess x for $\textsc{Opt}$ with $\textsc{Opt} \le x$ and where your goals is to find a feasible solution with cost at most $(1+\varepsilon)x$.

Assume that you want to find an approximate solution for a minimization problem with optimal objective value $\textsc{Opt}$. The optimal objective value itself is unknown to you, but you know that it is bounded by $T_{\text{low}} \le \textsc{Opt} \le T_{\text{upp}}$. The only tool that you have at your disposal is a black box (this black box is the PTAS for the special situation described above). If you feed the black box with an integer $x \ge \textsc{Opt}$, then the black box will give you a feasible solution with objective value at most $(1+\varepsilon)x$. If you feed the black box with an integer $x < \textsc{Opt}$, then the black box will return a random feasible solution.

How can you use this black to find a globally good approximate solution? How often will you use the black box in terms of $T_{\text{upp}}/T_{\text{low}}$? [Hint: Call an integer x a winner, if the black box returns a feasible solution with cost at most $(1+\varepsilon)x$, and call it a loser otherwise. Note that every loser x fulfills $x < \textsc{Opt}$. For $k \ge 0$, define $x_k = (1+\varepsilon)^k T_{\text{low}}$. Call the black box for the inputs x_k until you find a winner.]

0.5 Structuring the execution of an algorithm

As third standard approach to the construction of approximation schemes we discuss the technique of *adding structure to the execution of an algorithm.* Here the main idea is to take an exact but slow algorithm A, and to interact with it while it is working. If the algorithm accumulates a lot of auxiliary data during its execution, then we may remove part of this data and clean up the algorithm's memory. As a result the algorithm becomes faster (since there is less data to process) and generates an incorrect output (since the removal of data introduces errors). In the ideal case, the time complexity of the algorithm becomes polynomial and the incorrect output constitutes a good approximation of the true optimum.

This approach can only work out if the algorithm itself is highly structured. In this chapter we will only deal with rather primitive algorithms that do not even try to optimize something. They simply generate all feasible solutions and only suppress obvious duplicates. They are of a severely restricted form and work in a severely restricted environment. The following two Definitions 0.5.1 and 0.5.2 define the type of optimization problem X and the type of algorithm A to which this approach applies.

Definition 0.5.1 *(Properties of the optimization problem)*

Every instance I of the optimization problem X naturally decomposes into n pieces $P_1, \ldots, P_n$. For every k with $1 \le k \le n$, the prefix sequence $P_1, \ldots, P_k$ of pieces forms an instance of the optimization problem X. We write I_k short for this prefix sequence.

The relevant properties of every feasible solution σ of I can be concisely encoded by a d-dimensional vector $\vec{v}(\sigma)$ with non-negative coordinates such that the following two conditions are satisfied:

(i) The objective value of σ can be deduced in constant time from $\vec{v}(\sigma)$.

(ii) An algorithm (as described in Definition 0.5.2 below) is able to digest the vectors that encode the feasible solutions for I_k and to deduce from them and from the piece P_k the vectors for all feasible solutions of I_{k+1}.

This definition is so vague that it needs an immediate illustration. Consider once again the scheduling problem $P2\,|\,|\,C_{\max}$ from Sections 0.3.1 and 0.4.1. The job J_k ($k = 1, \ldots, n$) forms the input piece P_k, and the instance I_k is the restriction of instance I to the first k jobs. The concise encoding of a feasible schedule σ with machine loads L_1 and L_2 simply is the two-dimensional vector $\vec{v}(\sigma) = [L_1, L_2]$. And the objective value of σ can be deduced from $\vec{v}(\sigma)$ by taking the maximum of the two coordinates, as we required in condition (i) above.

Definition 0.5.2 *(Properties of the algorithm)*

(Initialization). Algorithm A computes in constant time all feasible solutions for the instance I_1 and puts the corresponding encoding vectors into the vector set VS_1.

(Phases). Algorithm A works in phases. In the k-th phase ($k = 2, \ldots, n$) the algorithm processes the input piece P_k and generates the vector set VS_k. *This is done as follows. The algorithm takes every possible vector $\vec{v}(\sigma)$ in* VS_{k-1} *and combines the input piece P_k in all possible ways with the underlying feasible solution σ. The vectors that encode the resulting feasible solutions form the set* VS_k. *We assume that every single vector in* VS_{k-1} *can be handled in constant time and only generates a constant number of vectors in* VS_k.

(Output). After the n-th phase, the algorithm computes for every vector in VS_n *the corresponding objective value and outputs the best solution that it finds.*

Note that two distinct feasible solutions σ_1 and σ_2 with $\vec{v}(\sigma_1) = \vec{v}(\sigma_2)$ are only stored once in any vector set. Now assume that X is an optimization problem as described in Definition 0.5.1 and that A is an algorithm for X as described in Definition 0.5.2. What can we say about the time complexity of A? The initialization step of A needs only constant time. Since every single vector in VS_{k-1} is handled in constant time, the time complexity for the k-th phase is proportional to $|\mathrm{VS}_{k-1}|$. The time complexity for the output step is proportional to $|\mathrm{VS}_n|$. To summarize, the overall time complexity of A is proportional to $\sum_{k=1}^{n} |\mathrm{VS}_k|$. In general this value $\sum_{k=1}^{n} |\mathrm{VS}_k|$ will be exponential in the size of the input, and hence algorithm A will not have polynomial time complexity.

Let us try to transform algorithm A into an approximation algorithm for problem X. Note that algorithm A accumulates a lot of intermediate data – all the vector sets VS_k – during its execution. We consider the vectors in VS_k as geometric points in d-dimensional Euclidean space. If two of these points are very close to each other (in the geometric sense) then they should also encode very similar feasible solutions. And in this case there is no reason to keep both of them! We simply remove one of them from the vector set. By doing many such removals in an approriate way, we may thin out the vector sets and (hopefully) bring their cardinalities down to something small. And if the cardinalities of all vector sets VS_k are small, then the time complexity of algorithm A is (polynomially) small, too. And as only feasible solutions were removed that had surviving feasible solutions close to them, we probably will get a fairly good approximate solution in the end.

The approach of adding structure to the execution of an algorithm was introduced by Ibarra & Kim [44] in 1975. Then Sahni [70] applied it to get an FPTAS for a variety of scheduling problems, and Gens & Levner [25] applied it to get an FPTAS for minimizing the weighted number of tardy jobs on a single machine. In the 1990s, the approach was for instance used in the papers by Potts & Van Wassenhove [68], Kovalyov, Potts & Van Wassenhove [52], and Kovalyov & Kubiak [51]. Woeginger [80] studied the applicability of this approach in a fairly general setting, and he identified several arithmetical and structural conditions on some underlying cost and transition functions that automatically guarantee the existence of an FPTAS. Since the conditions in [80] are very technical, we will not discuss them in this tutorial.

In the following four sections, we will illustrate the technique of adding structure to the execution of an algorithm with the help of four examples. Section 0.5.1 deals (once again!) with makespan minimization on two identical machines, Section 0.5.2 deals with the minimization of the total weighted job completion time on two identical machines, Section 0.5.3 gives an FPTAS for the knapsack problem, and Section 0.5.4 discusses the weighted number of tardy jobs on a single machine. Section 0.5.5 contains a number of exercises.

0.5.1 Makespan on two identical machines

The problem. We return once again to the problem $P2\,|\,|\,C_{\max}$ that was already discussed in Sections 0.3.1 and 0.4.1. Recall that in an instance I of $P2\,|\,|\,C_{\max}$ there are n jobs J_j ($j = 1, \ldots, n$) with processing times p_j, and the goal is to find a schedule on two identical machines M_1 and M_2 that minimizes the makespan. Denote $p_{\text{sum}} = \sum_{j=1}^{n} p_j$. Observe that the size $|I|$ of the input I satisfies

$$|I| \;\geq\; \log(p_{\text{sum}}) \;=\; \text{const} \cdot \ln(p_{\text{sum}}). \tag{0.12}$$

Just to write down all the values p_j in binary representation we need $\log(p_{\text{sum}})$ bits! We note that the exact value of the constant in (0.12) equals $\log(e)$ where e is the base of the natural logarithm. For our purposes it is sufficient to know that it is some positive number that does not depend on the instance I. In this section, we will find a third approximation scheme for $P2\,|\,|\,C_{\max}$. This time, we will find an FPTAS whereas the approximation schemes in Sections 0.3.1 and 0.4.1 only were PTAS's.

We encode a feasible schedule σ with machine loads L_1 and L_2 by the two-dimensional vector $[L_1, L_2]$. Condition (i) in Definition 0.5.1 is trivially fulfilled, since the objective value of the encoded schedule σ equals $\max\{L_1, L_2\}$. Condition (ii) goes hand in hand with the algorithm A from Definition 0.5.2.

> **Initialization.** Set $\mathrm{VS}_1 = \{[p_1, 0],\ [0, p_1]\}$.
>
> **Phase k.** For every vector $[x, y]$ in VS_{k-1}, put $[x + p_k, y]$ and $[x, y + p_k]$ in VS_k.
>
> **Output.** Output the vector $[x, y] \in \mathrm{VS}_n$ that minimizes the value $\max\{x, y\}$.

Since the coordinates of all vectors in all sets VS_k are integers in the range from 0 to p_{sum}, the cardinality of every vector set VS_k is bounded from above by $O(p_{\mathrm{sum}}^2)$. Since the time complexity of the algorithm is proportional to $\sum_{k=1}^{n} |\mathrm{VS}_k|$, the algorithm has a pseudo-polynomial time complexity of $O(n\, p_{\mathrm{sum}}^2)$.

How to simplify the vector sets. All considered vectors correspond to geometric points in the rectangle $[0, p_{\mathrm{sum}}] \times [0, p_{\mathrm{sum}}]$. We subdivide this rectangle with horizontal and vertical cuts into lots of boxes; in both directions these cuts are made at the coordinates Δ^i for $i = 1, 2, \ldots, L$ where

$$\Delta \;=\; 1 + \frac{\varepsilon}{2n}. \tag{0.13}$$

and where

$$L \;=\; \lceil \log_\Delta(p_{\mathrm{sum}}) \rceil \;=\; \lceil \ln(p_{\mathrm{sum}}) / \ln(\Delta) \rceil \;\le\; \lceil (1 + \frac{2n}{\varepsilon}) \ln(p_{\mathrm{sum}}) \rceil. \tag{0.14}$$

The last inequality holds since for all $z \ge 1$ we have $\ln z \ge (z-1)/z$ (which can be seen from the Taylor expansion of $\ln z$). If two vectors $[x_1, y_1]$ and $[x_2, y_2]$ fall into the same box of this subdivision, then their coordinates satisfy

$$x_1/\Delta \;\le\; x_2 \;\le\; x_1\,\Delta \qquad \text{and} \qquad y_1/\Delta \;\le\; y_2 \;\le\; y_1\,\Delta. \tag{0.15}$$

Since Δ is very small, vectors in the same box indeed are very close to each other. Now it is straightforward to simplify the vector set VS_k: Out of every box that has non-empty intersection with VS_k we select a single vector and put it into the so-called *trimmed* vector set $\mathrm{VS}_k^{\#}$. All remaining vectors from the vector set VS_k that have not been selected are lost for the further computations. And in phase $k+1$, the so-called *trimmed* algorithm generates its new vector set from the smaller set $\mathrm{VS}_k^{\#}$, and not from the set VS_k. What can be said about the resulting time complexity? The trimmed vector set $\mathrm{VS}_k^{\#}$ contains at most one vector from each box in the subdivision. Altogether there are $O(L^2)$ boxes, and so by (0.14) and by (0.12) the number of boxes is polynomial in the input size $|I|$ and also polynomial in $1/\varepsilon$. And since the time complexity of the trimmed algorithm is proportional to $\sum_{k=1}^{n} |\mathrm{VS}_k^{\#}|$, the trimmed algorithm has a time complexity that is polynomial in the input size and in $1/\varepsilon$.

How to analyze the worst case behavior. The original (untrimmed) algorithm works with vector sets $\mathrm{VS}_1, \ldots, \mathrm{VS}_n$ where VS_{k+1} is always computed from VS_k. The trimmed algorithm works with vector sets $\mathrm{VS}_1^\#, \ldots, \mathrm{VS}_n^\#$ where $\mathrm{VS}_{k+1}^\#$ is always computed from $\mathrm{VS}_k^\#$. We will prove the following statement by induction on k that essentially says that the vector set $\mathrm{VS}_k^\#$ is a decent approximation of the vector set VS_k: For every vector $[x, y] \in \mathrm{VS}_k$ there exists a vector $[x^\#, y^\#]$ in $\mathrm{VS}_k^\#$ such that

$$x^\# \;\le\; \Delta^k\, x \qquad \text{and} \qquad y^\# \;\le\; \Delta^k\, y. \tag{0.16}$$

Indeed, for $k = 1$ the statement follows from the inequalities in (0.15). Now assume that the statement holds true up to some index $k-1$, and consider an arbitrary vector $[x, y] \in \mathrm{VS}_k$. The untrimmed algorithm puts this vector into VS_k when it adds job J_k to some feasible schedule for the first $k-1$ jobs. This feasible schedule for the first $k-1$ jobs is encoded by a vector $[a, b] \in \mathrm{VS}_{k-1}$, and either $[x, y] = [a + p_k, b]$ or $[x, y] = [a, b + p_k]$ must hold. Since both cases are completely symmetric, we assume without loss of generality that $[x, y] = [a + p_k, b]$. By the inductive assumption, there exists a vector $[a^\#, b^\#]$ in $\mathrm{VS}_{k-1}^\#$ with

$$a^\# \;\le\; \Delta^{k-1}\, a \qquad \text{and} \qquad b^\# \;\le\; \Delta^{k-1}\, b. \tag{0.17}$$

The trimmed algorithm generates the vector $[a^\# + p_k, b^\#]$ in the k-th phase. The trimming step may remove this vector again, but it must leave some vector $[\alpha, \beta]$ in $\mathrm{VS}_k^\#$ that is in the same box as $[a^\# + p_k, b^\#]$. Now we are done. This vector $[\alpha, \beta]$ is an excellent approximation of $[x, y] \in \mathrm{VS}_k$ in the sense of (0.16). Indeed, its first coordinate α satisfies

$$\alpha \;\le\; \Delta(a^\# + p_k) \;\le\; \Delta^k\, a + \Delta\, p_k \;\le\; \Delta^k(a + p_k) \;=\; \Delta^k x.$$

For the first inequality, we used that $[\alpha, \beta]$ and $[a^\# + p_k, b^\#]$ are in the same box, and for the second inequality, we used (0.17). By analogous arguments, we can show that $\beta \le \Delta^k y$, and this completes the inductive proof. This proof looks somewhat technical, but its essence is very simple: The untrimmed algorithm and the trimmed algorithm roughly run in parallel and always perform roughly parallel actions on perturbed versions of the same data.

At the very end of its execution, the untrimmed algorithm outputs that vector $[x, y]$ in VS_n that minimizes the value $\max\{x, y\}$. By our inductive proof there is a vector $[x^\#, y^\#]$ in $\mathrm{VS}_n^\#$ whose coordinates are at most a factor of Δ^n above the corresponding coordinates of $[x, y]$. We conclude that

$$\max\{x^\#, y^\#\} \;\le\; \max\{\Delta^n\, x, \Delta^n\, y\} \;=\; \Delta^n \max\{x, y\} \;=\; \Delta^n \textsc{Opt},$$

and that our algorithm is a Δ^n-approximation algorithm for $P2\,|\,|\,C_{\max}$. How large is Δ^n? In (0.13) we defined $\Delta = 1 + \frac{\varepsilon}{2n}$. A well-known inequality says that $(1 + z/n)^n \le 1 + 2z$ holds for $0 \le z \le 1$ (The left-hand side of this inequality is a convex function in z, and its right-hand side is a linear function in z. Moreover, the inequality holds true at the two endpoints $z = 0$ and $z = 1$). By setting $z = \varepsilon/2$ in this inequality, we get that $\Delta^n \le 1+\varepsilon$. So we indeed have constructed an FPTAS!

Discussion. What are the special properties of problem $P2\,|\,|\,C_{\max}$ and of the untrimmed algorithm that make this approach go through? Well, to get the time complexity down to polynomial we made heavy use of the fact that the lengths of the sides of the rectangle $[0, p_{\text{sum}}] \times [0, p_{\text{sum}}]$ grow at most exponentially with the input size. The variable L defined in (0.14) depends logarithmically on these sidelengths, and the final time complexity of the trimmed algorithm depends polynomially on L. So, one necessary property is that the coordinates of all vectors in the vector sets are exponentially bounded in $|I|$. The number of boxes is roughly L^d where d is the dimension of the vectors. Hence, it is important that this dimension is a fixed constant and does not depend on the input.

The inductive argument in our analysis of the worst case behavior is based on fairly general ideas. One way of looking at the algorithm is that it translates old vectors for the instance I_{k-1} into new vectors for the instance I_k. The two translation functions are $F_1[x, y] = [x + p_k, y]$ and $F_2[x, y] = [x, y + p_k]$. The coordinates of the new vectors are non-negative linear combinations of the coordinates of the old vectors, and in fact that is the crucial property that makes the inductive argument go through; see Exercise 0.5.2 in Section 0.5.5.

In the final step, the algorithm extracts the optimal objective value from the final vector set. This corresponds to another function $G[x, y] = \max\{x, y\}$ that translates vectors into numbers. Here the crucial property is that the value $G[(1+\varepsilon)x, (1+\varepsilon)y]$ is always relatively close to the value $G[x, y]$. This property is crucial, but also fairly weak. It is fulfilled by most natural and by many unnatural objective functions. The reader may want to verify that for instance the function $G[x, y] = x^3 + \max\{4x, y^2\}$ also satisfies this property. For a more general discussion and a better view on this approach we refer to Woeginger [80].

0.5.2 Total weighted job completion time on two identical machines

The problem. In the scheduling problem $P2\,|\,|\,\sum w_j C_j$ the input consists of n jobs J_j ($j = 1, \ldots, n$) with positive integer processing times p_j and positive integer weights w_j. All jobs are available for processing at time 0, and the goal is to schedule them on two identical machines such that the weighted sum of job completion times is minimized. Denote $p_{\text{sum}} = \sum_{j=1}^{n} p_j$ and $w_{\text{sum}} = \sum_{j=1}^{n} w_j$, and observe that the size $|I|$ of the input I is at least $\log(p_{\text{sum}}) + \log(w_{\text{sum}})$. The problem $P2\,|\,|\,\sum w_j C_j$ is NP-hard in the ordinary sense (Bruno, Coffman & Sethi [16]). In this section we describe an FPTAS for $P2\,|\,|\,\sum w_j C_j$ that uses the technique of adding structure to the execution of an algorithm. We will roughly follow the ideas of Sahni [70].

The solution. We first renumber the jobs such that $p_1/w_1 \le p_2/w_2 \le \cdots \le p_n/w_n$ holds. A simple job interchange argument (see for instance [55]) shows that there always exists an optimal schedule which does not contain any idle time, and in which both machines process the jobs in increasing order of index. We encode a feasible schedule σ by a 3-dimensional vector $[L_1, L_2, Z]$: The first coordinate L_1 is the total load of σ on the first machine, the second coordinate L_2 is the total load on the second machine, and the third coordinate Z is the weighted sum of job completion times in σ. Our untrimmed algorithm A now looks as follows.

> **Initialization.** Set $\mathrm{VS}_1 = \{[p_1, 0, p_1w_1],\ [0, p_1, p_1w_1]\}$.
>
> **Phase k.** For every vector $[x, y, z]$ in VS_{k-1}, put the vectors $[x + p_k, y, z + w_k(x + p_k)]$ and $[x, y + p_k, z + w_k(y + p_k)]$ in VS_k.
>
> **Output.** Output the vector $[x, y, z] \in \mathrm{VS}_n$ that minimizes the z value.

In the k-th phase, the algorithm schedules job J_k at the end of the first machine which increases the machine load by p_k and the objective value by $w_k(x + p_k)$, and it schedules J_k at the end of the second machine which increases the machine load by p_k and the objective value by $w_k(y + p_k)$. If one compares this algorithm to the algorithm in Section 0.5.1 then one realizes that there are many similarities between the two of them. In fact, the definition of the trimmed algorithm and the analysis of its worst case behavior can be caried out almost analogously to Section 0.5.1. Therefore, we will only briefly indicate that all the properties are fulfilled that took shape in the discussion at the end of Section 0.5.1.

The first two coordinates of all vectors in all sets VS_k are integers in the range from 0 to p_{sum}, and the third coordinate is an integer in the range from 0 to $p_{\mathrm{sum}}w_{\mathrm{sum}}$. The coordinates of all vectors in the vector sets are exponentially bounded in $|I|$, exactly as we required in the discussion at the end of Section 0.5.1. There are two translation functions that translate old vectors for the instance I_{k-1} into new vectors for the instance I_k, and in both of them the coordinates of the new vectors are non-negative linear combinations of the coordinates of the old vectors. That is all we need for an inductive proof that the vector set $\mathrm{VS}_k^{\#}$ is a good approximation of the vector set VS_k, see Exercise 0.5.2 in Section 0.5.5. Finally, the objective value is extracted from a vector $[x, y, z]$ by the projection $G[x, y, z] = z$. If the values x, y, and z are perturbed by at most a factor of $1 + \varepsilon$, then the value of the projection is increased by at most a factor of $1 + \varepsilon$. Hence, we indeed have all the ingredients that are needed to make the approach go through. This yields the FPTAS.

0.5.3 The knapsack problem

The problem. In the 0/1-knapsack problem an instance I consists of n items with profits p_j and weights w_j $(j = 1, \ldots, n)$, together with a weight bound W. All numbers in the data are non-negative integers. The goal is to select a subset of the items whose total weight does not exceed the weight bound W such that the total profit of the selected items is maximized. Denote $p_{\rm sum} = \sum_{j=1}^{n} p_j$ and $w_{\rm sum} = \sum_{j=1}^{n} w_j$; the size $|I|$ is at least $\log(p_{\rm sum})+\log(w_{\rm sum})$. The 0/1-knapsack problem is NP-hard in the ordinary sense (Karp [48]). The first FPTAS for 0/1-knapsack is due to Ibarra & Kim [44]. We present an FPTAS for 0/1-knapsack that uses the technique of adding structure to the execution of an algorithm.

The solution. We encode a subset of the items by a two-dimensional vector $[\omega, \pi]$: The first coordinate ω is the total selected weight, and the second coordinate π is the total selected profit. Consider the following algorithm for the 0/1-knapsack problem in the framework of Definition 0.5.2.

> **Initialization.** If $w_1 \le W$ then set $\mathrm{VS}_1 = \{[w_1, p_1],\ [0,0]\}$. And if $w_1 > W$ then set $\mathrm{VS}_1 = \{[0,0]\}$.
>
> **Phase k.** For every vector $[x, y]$ in VS_{k-1} do the following. Put $[x, y]$ into VS_k. If $x + w_k \le W$ then put $[x + w_k, y + p_k]$ in VS_k.
>
> **Output.** Output the vector $[x, y] \in \mathrm{VS}_n$ that maximizes the y value.

This algorithm is based on the standard dynamic programming formulation of the 0/1-knapsack (Bellman & Dreyfus [15]). It looks fairly similar to the algorithms we discussed in Sections 0.5.1 and 0.5.2. However, there is a fundamental difference: If-statements. The algorithms in Sections 0.5.1 and 0.5.2 treated all vectors in all vector sets exactly the same way. They translated the old vectors *unconditionally* into new vectors. The above if-statements, however, treat different vectors in a different way, and that makes the computation of the vector sets less robust. But let us start with the properties that can be carried over to our situation without any effort. The first coordinates of all vectors are integers in the range from 0 to $w_{\rm sum}$, and the second coordinates are integers in the range from 0 to $p_{\rm sum}$. Hence, both coordinates are exponentially bounded in $|I|$, exactly as is needed. There are two translation functions that translate old vectors for the instance I_{k-1} into new vectors for the instance I_k. In both of them the coordinates of the new vectors are non-negative linear combinations of the coordinates of the old vectors, exactly as we need. The objective value is extracted from a vector $[x, y]$ by the projection $G[x, y] = y$. If the values x and y are perturbed by at most a factor of $1 + \varepsilon$, then the value of the projection is increased by at most a factor of $1 + \varepsilon$, exactly as we need. So — it seems that we have everything exactly as we need to make the approach go through. Where is the problem?

The problem is that the inductive proof showing that $\mathrm{VS}_k^\#$ is a good approximation of VS_k does not go through. Read the inductive argument in Section 0.5.1 again. In a nutshell, this inductive argument finds for any vector $\vec{v}$ in VS_k a Δ^k-approximation $\vec{v}^\# \in \mathrm{VS}_k^\#$ in the following way. The vector $\vec{v}$ results from a vector $\vec{u} \in \mathrm{VS}_k$ by a certain translation τ. By the inductive assumption, this vector $\vec{u}$ has a Δ^{k-1}-approximation $\vec{u}^\# \in \mathrm{VS}_{k-1}^\#$ which can be translated by the same translation τ into a vector $\vec{z}^{\#}$. In the end, the vector that is selected from the same box as $\vec{z}^{\#}$ for $\mathrm{VS}_k^{\#}$ yields the desired Δ^k-approximation for $\vec{v}$. Done. The trouble, however, is that the vector $\vec{z}^{\#}$ itself might be infeasible. Then the if-statement weeds out $\vec{z}^{\#}$ immediately and the argument breaks down. We stress that this problem is not only a technicality of the proof, but in fact it may cause disastrous worst case behavior; see Exercise 0.5.9 in Section 0.5.5. How can we rescue our approach? All the trouble originates from infeasibilities. So, during the trimming maybe we should not take an arbitrary vector from every box, but should always select the vector that is furthest from causing infeasibility? And the vectors that are furthest from causing infeasibility are of course the vectors with the smallest x-coordinates! This indeed works out. The new trimming step becomes the following:

> From every box that intersects the untrimmed vector set, select a single vector with minimum x-coordinate and put it into the trimmed vector set $\mathrm{VS}_k^{\#}$.

And the new inductive statement becomes the following:

> For every vector $[x, y]$ in the vector set VS_k of the untrimmed algorithm, there exists a vector $[x^{\#}, y^{\#}]$ in $\mathrm{VS}_k^{\#}$ such that $x^{\#} \leq x$ and $y^{\#} \leq \Delta^k\, y$.

Note that the main difference to the old inductive statement in (0.16) is that now we have the strong inequality $x^{\#} \leq x$ instead of the weaker $x^{\#} \leq \Delta^k\, x$. The inductive proof is fairly easy and is left to the reader, see Exercise 0.5.10 in Section 0.5.5.

Discussion. What are the special properties of the knapsack problem that make this modified approach go through? As a rule of thumb, the approach goes through as long as the region of feasible vectors $[x_1, \ldots, x_d]$ forms a halfspace of $\mathbb{R}^d$ whose bounding hyperplane is of the form $\sum_{i=1}^{d} a_i x_i \leq A$ with non-negative real numbers a_i and A. For instance in the knapsack problem above, the feasible region is specified by the inequality $x \leq W - w_k$ which clearly satisfies this rule of thumb. The bounding hyperplane guides us in selecting the vectors that are furthest from causing infeasibility: From every box we select the vector that is furthest from the hyperplane. And this often makes the inductive proof go through (we note also that the non-negativity of the coefficients a_i of the hyperplane is usually needed in the inductive argument). Exercise 0.5.11 in Section 0.5.5 gives an illustration for this.

On the other hand, as soon as the feasible region is of a more complicated form, then in general there is no hope of getting the approach to go through. There is no simple way to identify the most promising feasible solution in a box. Consider for instance vectors $[x, y]$ in two-dimensional space whose feasible region is specified by $x \leq 20$ and $y \leq 20$. If a box contains the two vectors $[9, 11]$ and $[10, 10]$, which of them shall we put into the trimmed vector set? Either choice may lead to a catastrophy.

0.5.4 Weighted number of tardy jobs on a single machine

The problem. In the scheduling problem $1\,|\,|\,\sum w_j U_j$ the input consists of n jobs J_j ($j = 1, \ldots, n$) with processing times p_j, weights w_j, and due dates d_j. All jobs are available for processing at time 0. In some schedule a job is called early if its processing is completed by its deadline, and otherwise it is called tardy. The goal is to schedule the jobs on a single machine such that the total weight of the tardy jobs is minimized. Denote $p_{\text{sum}} = \sum_{j=1}^{n} p_j$ and $w_{\text{sum}} = \sum_{j=1}^{n} w_j$; the size $|I|$ is at least $\log(p_{\text{sum}}) + \log(w_{\text{sum}})$. Problem $1\,|\,|\,\sum w_j U_j$ is NP-hard in the ordinary sense (Karp [48]). The first FPTAS for $1\,|\,|\,\sum w_j U_j$ is due to Gens & Levner [25]. We present an FPTAS for $1\,|\,|\,\sum w_j U_j$ that uses the technique of adding structure to the execution of an algorithm.

The solution. We renumber the jobs such that $d_1 \leq d_2 \leq \cdots \leq d_n$. Under this numbering, there always exists an optimal schedule in which all early jobs are processed before all tardy jobs and in which all early jobs are processed in increasing order of index. We encode feasible schedules by two-dimensional vectors $[P, W]$: The first coordinate P is the total processing time of the scheduled early jobs, and the second coordinate W is the total weight of the tardy jobs (that all are shifted far to the future). Consider the following algorithm for $1\,|\,|\,\sum w_j U_j$ in the framework of Definition 0.5.2.

> **Initialization.** If $p_1 \leq d_1$ then set $\mathrm{VS}_1 = \{[p_1, 0],\ [0, w_1]\}$. And if $p_1 > d_1$ then set $\mathrm{VS}_1 = \{[0, w_1]\}$.
>
> **Phase k.** For every vector $[x, y]$ in VS_{k-1} do the following. Put $[x, y + w_k]$ into VS_k. If $x + p_k \leq d_k$ then put $[x + p_k, y]$ in VS_k.
>
> **Output.** Output the vector $[x, y] \in \mathrm{VS}_n$ that minimizes the y value.

This algorithm is based on the dynamic programming formulation of Lawler & Moore [56]. In the k-th phase, the algorithm first schedules job J_k tardy which increases the objective value y by w_k. Then it tries to schedule J_k early which increases the total processing time of the early jobs to $x + p_k$ and only is possible if this value does not exceed the due date d_k.

The first coordinates of the vectors are bounded by p_{sum}, and their second coordinates are bounded by w_{sum}. In the two translation functions that translate old vectors for the instance I_{k-1} into new vectors for the instance I_k, the coordinates of the new vectors are non-negative linear combinations of the coordinates of the old vectors. The objective value is extracted from a vector $[x, y]$ by the same projection $G[x, y] = y$ as in the knapsack algorithm in Section 0.5.3. What about if-statements and infeasibilities? Only the x-coordinate can cause infeasibilities, and so we may always select a vector with minimum x-coordinate from any box and put it into the trimmed vector set $\mathrm{VS}_k^{\#}$. Hence, the same machinery as in the preceding sections and the same inductive proof as in Section 0.5.3 will yield the FPTAS.

0.5.5 Exercises

Exercise 0.5.1. When we discussed the general framework at the beginning of Section 0.5, we required in Definition 0.5.1 that extracting the corresponding objective value from a vector only takes *constant* time. Moreover, in Definition 0.5.2 we required that the initialization can be done in *constant* time, and we required that every fixed vector in VS_{k-1} only generates a *constant* number of vectors in VS_k.

Would the approach break down, if we replace the word 'constant' by the word 'polynomial'? Are these restrictions necessary at all?

Exercise 0.5.2. Let X be an optimization problem X as described in Definition 0.5.1, and let A be an algorithm for X as described in Definition 0.5.2. The feasible solutions are encoded by d-dimensional vectors. The algorithm generates the vector set VS_k by applying the functions $F_1, \ldots, F_q$ to every vector in VS_{k-1}. Every F_i $(1 \le i \le q)$ is a linear function from $\mathbb{R}^d \to \mathbb{R}^d$, and all the coordinates of the images are non-negative linear combinations of the coordinates of the preimages. The coefficients of these non-negative linear combinations may depend on the input piece P_k, but they must remain non-negative. We perform the trimming of the vector sets as in Section 0.5.1.

Prove that under these conditions, for every vector $\vec{v}$ in VS_k there exists a vector $\vec{v}^{\#}$ in $\mathrm{VS}_k^{\#}$ whose coordinates are at most a factor of Δ^k above the corresponding coordinates of $\vec{v}$. [Hint: Translate the proof from Section 0.5.1 to the more general situation.]

Exercise 0.5.3. Construct an FPTAS for $Pm\,|\,|\,C_{\max}$ by appropriately modifying the approach described in Section 0.5.1. How do you encode the feasible schedules as vectors? What is the dimension of these vectors? How does the worst case guarantee depend on ε and m? How does the time complexity depend on ε and m? Can you extend this approach to get an approximation scheme for $P\,|\,|\,C_{\max}$?

Exercise 0.5.4. Consider n jobs J_j $(j = 1, \ldots, n)$ with positive integer processing times p_j on three identical machines. The goal is to find a schedule with machine loads L_1, L_2, and L_3 that minimizes the value $L_1^2 + L_2^2 + L_3^2$, i.e., the sum of squared machine loads.

Construct an FPTAS for this problem by following the approach described in Section 0.5.1. Can you generalize this FPTAS to any fixed number m of machines?

Exercise 0.5.5. Construct an FPTAS for $R2\,|\,|\,C_{\max}$, i.e., makespan minimization on two unrelated machines (see Section 0.4.2). What are the main differences between this FPTAS and the FPTAS for makespan minimization on two identical machines described in Section 0.5.1? Can you extend this approach to get an FPTAS for $Rm\,|\,|\,C_{\max}$?

Exercise 0.5.6. Generalize the FPTAS in Section 0.5.2 to $Pm\,||\,\sum w_jC_j$ with a fixed number of machines. How does the dimension of the vectors that encode the schedules depend on m? Can you extend the approach to $P\,||\,\sum w_jC_j$? Can you extend the approach to $R2\,||\,\sum w_jC_j$? And to $Rm\,||\,\sum w_jC_j$? What about $R\,||\,\sum w_jC_j$?

Exercise 0.5.7. Consider n jobs J_j ($J = 1,\ldots,n$) with processing times p_j on two identical machines. The goal is to find a schedule that minimizes the sum of squared job completion times. Design an FPTAS for this problem by modifying the arguments from Section 0.5.2. Can you also handle the variant where the goal is to minimize the sum of cubed job completion times?

Exercise 0.5.8. Section 0.5.2 discussed the problem of minimizing the sum of weighted job *completion* times on two identical machines. This exercise asks you to find an FPTAS for the problem of minimizing the sum of weighted job *starting* times on two identical machines. How do you order the jobs in the beginning?

Exercise 0.5.9. Assume that we apply the technique of adding structure to the execution of an algorithm to the knapsack problem as discussed in Section 0.5.3. Assume that in the trimming we select an *arbitrary* vector from every box and do not care about critical coordinates.

Construct knapsack instances for which the worst case ratio of this approach may become arbitrarily close to 2. [Hint: Use three items that all have roughly the same profit. Choose the weight w_1 very close to the weight bound W, and choose the weight w_2 even closer to W. Choose w_3 such that the first and third item together from a feasible solution, whereas the second and third item together are not feasible. Make the trimming select the 'wrong' vector for $\mathrm{VS}_2^{\#}$.]

Exercise 0.5.10. In Section 0.5.3 we designed a new trimming step for the knapsack problem that from every box selects a vector with minimum x-coordinate for $\mathrm{VS}_k^{\#}$. Prove that the resulting algorithm throughout fulfills the following statement: For every vector $[x, y] \in \mathrm{VS}_k$ there is a vector $[x^{\#}, y^{\#}] \in \mathrm{VS}_k^{\#}$ with $x^{\#} \le x$ and $y^{\#} \le \Delta^k y$. [Hint: Follow the inductive argument in Section 0.5.1.]

Exercise 0.5.11. Consider non-negative integers a_j and b_j ($j = 1,\ldots,n$), and a non-negative integer C. For a subset $K \subseteq \{1,\ldots,n\}$ denote $A_K = \sum_{j\in K} a_j$ and $B_K = \sum_{j\in K} b_j$. The goal is to find an index set K with $A_K + B_K \le C$ such that $A_K^2 + B_K^2$ is maximized.

Design an FPTAS for this problem. Encode a feasible subset K by the vector $[A_K, B_K]$, and use the approach that is implicit in the discussion at the end of Section 0.5.3.

Exercise 0.5.12. Consider n items with profits p_j and weights w_j ($j = 1,\ldots,n$), together with a weight bound W. The goal is to fill *two* knapsacks of capacity W with the items such that the total profit of the packed items is maximized.

Can you get an FPTAS for this problem by modifying the approach in Section 0.5.3? What is the main obstacle? Check your thoughts against Exercise 0.6.4 in Section 0.6.5.

Exercise 0.5.13. In Section 0.5.4 we described an FPTAS for minimizing the total weight of the tardy jobs on a single machine. This exercise deals with the dual problem of maximizing the total weight of the early jobs on a single machine. We use the notation from Section 0.5.4.

(a) Suppose that we call the FPTAS from Section 0.5.4 for the minimization problem, and then use the resulting schedule as an approximation for the maximization problem. Explain why this will not yield an FPTAS for the maximization problem.

(b) Design an FPTAS for maximizing the total weight of the early jobs on a single machine.

Exercise 0.5.14. This exercise asks you to construct an FPTAS for the following scheduling problem $1\,|\,|\,\mathrm{Rej}+\sum C_j$ with job rejections. An instance consists of n jobs J_j $(j = 1, \ldots, n)$ with a rejection penalty e_j and a processing time p_j. The goal is to reject a subset of the jobs and to schedule the remaining jobs on a single machine, such that the total penalty of all rejected jobs plus the sum of completion times of the scheduled jobs becomes minimum. This problem is NP-hard in the ordinary sense (Engels, Karger, Kolliopoulos, Sengupta, Uma & Wein [19]).

How do you get an FPTAS via the technique of adding structure to the execution of an algorithm? How do you encode the feasible schedules as vectors? What is the dimension of these vectors? [Hint: The problem variant without rejections is solved to optimality by scheduling the jobs in the order of non-decreasing lengths. This is a special case of Smith's ratio rule [75]. It is a good idea to first bring the jobs into this ordering, and then to go through them and to decide about rejections one by one. Then you only need to know the current total size of all scheduled jobs and the current objective value.]

0.6 How to get negative results

In the previous sections we have concentrated on positive results, and we have learnt how to construct approximation schemes. In this section we will focus on negative results, that is, we discuss in-approximability techniques for showing that a specific problem does not have a PTAS or an FPTAS unless P=NP. We remind the reader that throughout this chapter we only consider optimization problems in which all feasible solutions have non-negative cost.

Assume that we want to prove that some optimization problem X is hard to approximate. The first step in proving an in-approximability result always is to check through the web-compendium of Crescenzi & Kann [17] that contains an enormous list of references and results on in-approximability. At least, we will find out whether problem X is already known to be hard to approximate. If we do not find problem X in the compendium, then we will have to find our own in-approximability proof. The structure of an in-approximability proof usually is along the following line of argument:

> Suppose that there is a good approximability result for problem X. Then some intermediate problem Z_1 must have a good algorithm. Then some intermediate problem Z_2 must have a good algorithm .... Then some intermediate problem Z_{k-1} must have a good algorithm. Then the problem Z_k that is already known to be hard must have a good algorithm. Then P=NP.

The intermediate steps usually consist of classical polynomial time reductions between decision problems or by approximation preserving reductions between optimization problems. The number of intermediate problems $Z_1, \ldots, Z_k$ may vary, but the goal is to eventually reach the conclusion P=NP. Since P=NP is very unlikely to be true, it is also very unlikely that the starting assumption on the good approximability result for problem X is true. Hence, if in this section we talk about in-approximability and non-existence of some good algorithm, then we will always mean in-approximability and non-existence of some good algorithm *under the assumption that P≠NP.*

In the following four sections, we will illustrate some standard techniques for proving in-approximability results. Section 0.6.1 deals with disproving the existence of an FPTAS. Here the intermediate problem Z_1 is a strongly NP-hard problem, and in the first intermediate step we deduce the existence of a pseudo-polynomial algorithm for Z_1. Section 0.6.2 deals with disproving the existence of a PTAS by applying the gap technique. Here the intermediate problem Z_1 is an NP-hard problem, and in the first intermediate step we deduce the existence of a polynomial algorithm for Z_1. Section 0.6.3 deals with disproving the existence of a PTAS via APX-hardness arguments. Here the intermediate problems $Z_1, \ldots, Z_k$ are already known to be APX-hard, and the intermediate steps are approximation-preserving reductions from problem Z_{i+1} to problem Z_i. To prove the in-approximability of X, one only needs to establish an L-reduction from Z_1 to X; the remaining intermediate steps are proved in the literature. Section 0.6.4 discusses an example for an L-reduction. Finally, Section 0.6.5 contains a long list of exercises.

0.6.1 How to show that there is no FPTAS

The main tool for disproving the existence of an FPTAS for some optimization problem is to establish its strong NP-hardness. Strongly NP-hard problems (that fulfill some weak and natural supplementary condition) cannot have an FPTAS unless P=NP, or equivalently, every well-behaved problem that has an

FPTAS is solvable in pseudo-polynomial time. This is a surprising connection between seemingly unrelated concepts, and we want to discuss it in some detail. Let us start by considering the specific example $P\,|\,|\,C_{\max}$; we have studied this problem already in Section 0.3.2 where we designed a PTAS for it.

Example 0.6.1 *An instance I of $P\,|\,|\,C_{\max}$ is specified by the positive integer processing times p_j $(j = 1, \ldots, n)$ of n jobs together with the number m of identical machines. The goal is to find a schedule that minimizes the makespan. This problem is NP-hard in the strong sense (Garey & Johnson [24]), and therefore the problem remains NP-hard even if all numbers in the input are encoded in unary.*

One possible way of encoding an instance of $P\,|\,|\,C_{\max}$ in unary is to first write the value m in unary followed by the symbol #, followed by the list of job lengths in unary with the symbol \$ as a separation marker. Denote by $p_{sum} = \sum_{j=1}^{n} p_j$ the total job processing time. Then the length $|I|_{unary}$ of an instance I under this unary encoding equals $p_{sum} + m + n$. The instance I that consists of $m = 3$ machines together with six jobs with lengths $p_1 = 1$, $p_2 = 5$, $p_3 = 2$, $p_4 = 6$, $p_5 = 1$, $p_6 = 8$ (these numbers are written in decimal representation) is then encoded as the string 111#1\$11111\$11\$111111\$1\$11111111.

Throughout the next few paragraphs we will use the notation $|I|_{\text{unary}}$ and $|I|_{\text{binary}}$ so that we can clearly distinguish between unary and binary encodings of an instance. Now consider problem $P\,|\,|\,C_{\max}$ in the standard binary encoding where the size of instance I is $|I|_{\text{binary}}$, and suppose that this problem has an FPTAS. The time complexity of this FPTAS is bounded from above by an expression $|I|^{s}_{\text{binary}}/\varepsilon^{t}$ where s and t are fixed positive integers. Consider the algorithm A for $P\,|\,|\,C_{\max}$ that results from calling the FPTAS for an instance I with precision $\varepsilon = 1/(p_{\text{sum}} + 1)$. What is the time complexity of this algorithm, and how close does algorithm A come to the optimal objective value? The time complexity of algorithm A is bounded from above by $|I|^{s}_{\text{binary}} \cdot (p_{\text{sum}} + 1)^{t}$ which in turn is bounded by $O(|I|^{s+t}_{\text{unary}})$. Hence, A is a pseudo-polynomial algorithm for $P\,|\,|\,C_{\max}$. Next we claim that algorithm A always finds an optimal solution for instance I. Otherwise, we would have

$$\textsc{Opt}(I) + 1 \;\leq\; A(I) \;\leq\; (1 + \varepsilon)\textsc{Opt}(I) \;=\; (1 + \frac{1}{p_{\text{sum}} + 1})\textsc{Opt}(I).$$

Here the first inequality follows from the fact that all possible objective values are integers, and the other inequality follows by the chosen precision of approximation. The displayed inequality implies $\textsc{Opt}(I) \geq p_{\text{sum}} + 1$ which is blatantly wrong; even the trivial schedule that assigns all the jobs to the same machine has makespan at most p_{sum}. Hence, algorithm A outputs an optimal solution. To summarize, if there is an FPTAS for $P\,|\,|\,C_{\max}$ then there is a pseudo-polynomial algorithm A for $P\,|\,|\,C_{\max}$. And since $P\,|\,|\,C_{\max}$ is strongly NP-hard, the existence of a pseudo-polynomial algorithm for this problem implies P=NP. In other words, unless P=NP problem $P\,|\,|\,C_{\max}$ cannot have an FPTAS.

The above argument is fairly general. It only exploits the fact that all possible objective values are integral, and the fact that $\textsc{Opt} \le p_{\text{sum}}$. The argument easily generalizes to the following well-known theorem of Garey & Johnson [23]; see Exercise 0.6.1 in Section 0.6.5 for a sketch of its proof.

Theorem 0.6.2 *(Well-behaved and strongly NP-hard $\Longrightarrow$ no FPTAS)*
Let X be a strongly NP-hard minimization or maximization problem that satisfies the following two conditions:

(i) All feasible solutions of all instances I have integral costs.

(ii) There exists a polynomial p such that $\textsc{Opt}(I) \le p(|I|_{unary})$ holds for all instances I.

Then the optimization problem X does not have an FPTAS unless P=NP.

Note that the conditions (i) and (ii) in this theorem are not very restrictive, and are fulfilled by most natural optimization problems (and see Exercise 0.6.2 in Section 0.6.5 for some optimization problems that are less natural and do not fulfill these conditions). In other words, every well-behaved problem with an FPTAS is pseudo-polynomially solvable! What about the reverse statement? Does every well-behaved pseudo-polynomially solvable problem have an FPTAS? The answer to this question is no, and it actually is quite easy to construct (somewhat artificial) counter-examples that are pseudo-polynomially solvable problems but do not have an FPTAS; see for example Exercise 0.6.3 in Section 0.6.5. What we are going to do next is to discuss a quite natural counter-example with these properties.

Example 0.6.3 *(Korte & Schrader [50])*
Consider the following two-dimensional variant of the knapsack problem: There are n items with positive integer weights w_j and positive integer volumes v_j $(j = 1, \ldots, n)$, and there is a knapsack with a weight capacity W and a volume capacity V. The goal is to pack the maximum number of items into the knapsack without exceeding the weight and volume limits.

Although this two-dimensional knapsack problem is easily solved in pseudo-polynomial time, it does not have an FPTAS unless P=NP.

The proof is done by a reduction from the following partition problem that is NP-hard in the ordinary sense: The input to the partition problem consists of $2m$ positive integers $a_1, \ldots, a_{2m}$ that sum up to $2A$ and that fulfill $A/(m+1) < a_k < A/(m-1)$ for $k = 1, \ldots, 2m$. The problem is to decide whether there exists an index set K such that $\sum_{k \in K} a_k = A$ holds. From an instance of the partition problem, we now define the following instance of the knapsack problem with $n = 2m$ items: For $k = 1, \ldots, 2m$, item k has weight $w_k = a_k$ and volume $v_k = A - a_k$. The weight bound is $W = A$, and the volume bound is $V = (m-1)A$.

Now suppose that the two-dimensional knapsack problem does possess an FPTAS. Set $\varepsilon = \frac{1}{2m}$ and call the FPTAS for the constructed two-dimensional knapsack instance. Note that the resulting time complexity is polynomially bounded in the size of the two-dimensional knapsack instance, and that it is also polynomially bounded in the size of the partition instance. (1) First assume that the FPTAS returns a solution K with $|K| \geq m$ items. Denote $Z = \sum_{k \in K} a_k$. Then the restriction imposed by the weight capacity becomes $Z \leq A$, and the restriction imposed by the volume capacity becomes $|K|A - Z \leq (m-1)A$. Since $|K| \geq m$, this altogether implies $A \geq Z \geq (|K| - m + 1)A \geq A$. Consequently, the index set K constitutes a solution to the partition problem. (2) Now assume that the partition problem possesses a solution K with $\sum_{k \in K} a_k = A$. Since $A/(m+1) < a_k < A/(m-1)$ holds for all k, this yields $|K| = m$. Then the index set K constitutes a feasible solution to the knapsack problem with $|K| = m$ items. By the choice of ε, the approximate objective value that is computed by the FPTAS must be at least $(1 - \varepsilon)m > m - 1$. And since the objective function only takes integer values, the approximate objective value must be at least m. To summarize, the FPTAS would find in polynomial time a solution with at least m items for the knapsack problem if and only if the partition problem has a solution. Since the partition problem is NP-hard, this would imply P=NP.

Also this time our argument is fairly general. It only exploits the integrality of the objective value and the inequality $\textsc{Opt} \leq 2m$. It can be generalized to a proof for the following theorem; see Exercise 0.6.1 in Section 0.6.5.

Theorem 0.6.4 *(Very well-behaved and NP-hard $\Longrightarrow$ no FPTAS)*
Let X be an NP-hard minimization or maximization problem that satisfies the following two conditions:

(i) All feasible solutions of all instances I have integer cost.

(ii) There exists a polynomial p such that $\textsc{Opt}(I) \leq p(|I|)$ holds for all instances I.

Then the optimization problem X does not have an FPTAS unless P=NP.

0.6.2 How to show that there is no PTAS: The gap technique

The oldest and simplest tool for disproving the existence of a PTAS under the assumption that P≠NP is the so-called gap technique. The gap technique has first been used in the mid-1970s by Sahni & Gonzalez [71], Garey & Johnson [22], and Lenstra & Rinnooy Kan [58]. The idea is to do an NP-hardness proof via a polynomial time reduction that creates a wide gap between the objective values of NO-instances and the objective values of YES-instances. A more precise statement of this idea yields the following theorem.

Theorem 0.6.5 *(The gap technique)*
Let X be an NP-hard decision problem, let Y be a minimization problem, and let τ be a polynomial time computable transformation from the set of instances

of X into the set of instances of Y that satisfies the following two conditions for fixed integers $a < b$:

(i) Every YES-instance of X is mapped into an instance of Y with optimal objective value at most a.

(ii) Every NO-instance of X is mapped into an instance of Y with optimal objective value at least b.

Then problem Y does not have a polynomial time ρ-approximation algorithm with worst case ratio $\rho < b/a$ unless P=NP. Especially, problem Y does not have a PTAS unless P=NP.

There is an analogous theorem for maximization problems; see Exercise 0.6.5 in Section 0.6.5. Why is Theorem 0.6.5 true? Well, suppose that there would exist a polynomial time approximation algorithm whose worst case ratio ρ is strictly smaller than b/a. Take an arbitrary instance I of problem X, transform it in polynomial time with the help of τ, and feed the resulting instance $\tau(I)$ into the polynomial time approximation algorithm for problem Y. If I is a YES-instance, then the optimal objective value of $\tau(I)$ is at most a, and the approximation algorithm yields a solution with value strictly less than $(b/a)a = b$. If I is a NO-instance, then the optimal objective value of $\tau(I)$ is at least b, and the approximation algorithm cannot yield a solution with value better than the optimal value b. Hence, we could distinguish in polynomial time between YES-instances (the approximate value of $\tau(I)$ is $< b$) and NO-instances (the approximate value of $\tau(I)$ is $\geq b$) of the NP-hard problem X. And this of course would imply P=NP.

Theorem 0.6.6 *(Lenstra's impossibility theorem)*
Let Y be a minimization problem for which all feasible solutions of all instances I have integer cost. Let g be a fixed integer. Assume that the problem of deciding whether an instance I of Y has a feasible solution with cost at most g is NP-hard.

Then problem Y does not have a polynomial time ρ-approximation algorithm with worst case ratio $\rho < (g+1)/g$ unless P=NP. In fact, problem Y does not have a PTAS unless P=NP.

Lenstra's impossibility theorem is a special case of Theorem 0.6.5: As decision problem X, we use the problem of deciding whether there is a feasible solution with cost at most g. As polynomial time transformation τ from X to Y, we use the identity transformation. Moreover, we set $a = g$ and $b = g + 1$.

The two Theorems 0.6.5 and 0.6.6 are surprisingly simple and surprisingly useful. Let us give a demonstration of their applicability.

Example 0.6.7 *(Lenstra, Shmoys & Tardos [60])*
In the scheduling problem $R\,|\,|\,C_{\max}$ the input consists of m unrelated machines together with n jobs J_j $(j = 1, \ldots, n)$. If job J_j is assigned to machine M_i then its processing time equals p_{ij}. The goal is to find a schedule that minimizes the makespan.

The problem $R \,|\,|\, C_{\max}$ does not have a polynomial time ρ-approximation algorithm with worst case ratio $\rho < 4/3$ unless P=NP.

This result is established via Lenstra's impossibility theorem. More precisely, we will describe a polynomial time reduction from the well-known 3-Dimensional Matching problem (3DM); see Karp [48]. An instance of 3DM consists of three sets $A = \{a_1, \ldots, a_q\}$, $B = \{b_1, \ldots, b_q\}$, and $C = \{c_1, \ldots, c_q\}$, together with a subset T of $A \times B \times C$. The question is to decide whether T does contain a perfect matching T', that is, a subset T' of cardinality q that covers every element in $A \cup B \cup C$. Given any instance of 3DM, we now construct an instance of the scheduling problem $R \,|\,|\, C_{\max}$. There are $m = |T|$ machines where every machine corresponds to a triple in T, and the number n of jobs is equal to $|T| + 2q$. We distinguish two types of jobs: For every a_j, b_j, and c_j $(j = 1, \ldots, q)$ there are corresponding element jobs $J(a_j)$, $J(b_j)$, and $J(c_j)$. Moreover there are $|T| - q$ so-called dummy jobs. Now let us define the processing times of the jobs. Consider a machine M_i, and let $T_i = (a_j, b_k, c_l)$ be the triple that corresponds to M_i. On machine M_i, the processing time of the three element jobs $J(a_j)$, $J(b_k)$, and $J(c_l)$ equals 1. All the other jobs (i.e., all dummy jobs and the remaining $3q - 3$ element jobs) have processing time 3 on this machine. This completes the description of the scheduling instance.

(1) If T contains a perfect matching, then we can easily construct a schedule of length 3: For each triple $T_i = (a_j, b_k, c_l)$ in the perfect matching we schedule the three element jobs $J(a_j)$, $J(b_k)$, and $J(c_l)$ on machine M_i. We schedule the dummy jobs on the remaining empty machines. Hence, if we have a YES-instance of the 3DM problem, then there exists a schedule of length 3. (2) On the other hand, if there is a schedule of length 3, then there is a perfect matching. In a schedule of length 3 all element jobs must have length 1. Furthermore, there are $|T| - q$ machines reserved for the $|T| - q$ dummy jobs; the q remaining machines each process exactly three element jobs. It is easy to see that the triples corresponding to these q machines form a perfect matching. Hence, if we have a NO-instance of the 3DM problem, then (due to the integrality of job data) every feasible schedule has makespan at least 4.

Finally, we can apply Lenstra's impossibility theorem with $g = 3$ and thus get the 4/3 in-approximability result as claimed in the statement in Example 0.6.7. We remark that Lenstra, Shmoys & Tardos [60] even prove NP-hardness of deciding whether the makespan is at most $g = 2$; this of course leads to a better in-approximability bound of 3/2. We conclude this section by listing several in-approximability results in scheduling that have been derived via the gap technique. All listed results hold under the assumption P≠NP, and we are not going to repeat this assumption in every single case.

- Lenstra & Rinnooy Kan [58] prove that the problem $P \,|\, prec, p_j{=}1 C_{\max}$, i.e., makespan minimization on parallel identical machines with precedence constraints and unit processing times, does not have a polynomial time approximation algorithm with performance guarantee $\rho < 4/3$. The reduction is from the maximum clique problem in graphs and uses Theorem 0.6.6 with parameter $g = 3$.

- Hoogeveen, Lenstra & Veltman [39] derive in-approximability results for minimizing the makespan for jobs with unit processing times under communication delays. The variant with a restricted number of machines does not allow a performance guarantee $\rho < 4/3$ (Theorem 0.6.6 with parameter $g = 3$), whereas the variant with an unlimited number of machines does not allow a performance guarantee $\rho < 7/6$ (Theorem 0.6.6 with parameter $g = 6$).

- Williamson et al. [79] prove that the three shop problems $O\,|\,|\,C_{\max}$, $F\,|\,|\,C_{\max}$, and $J\,|\,|\,C_{\max}$ all do not allow polynomial time approximation algorithms with performance guarantee $\rho < 5/4$ (Theorem 0.6.6 with parameter $g = 4$).

- Kellerer, Tautenhahn & Woeginger [49] discuss the problem $1\,|\,r_j\,|\,\sum F_j$, i.e., minimizing the total flow time of n jobs on a single machine (where $F_j = C_j - r_j$). They show that no polynomial time approximation algorithm can have a performance guarantee $n^{1/2-\delta}$. This is done by using a variation of Theorem 0.6.5 with parameters $a \approx n^{1/2}$ and $b \approx n$. Leonardi & Raz [61] derive similar results for $P\,|\,r_j\,|\,\sum F_j$, i.e., minimization of the total flow time on parallel machines.

- Schuurman & Woeginger [73] show that makespan minimization of pipelined operator graphs does not allow a performance guarantee $\rho < 4/3$ by applying Theorem 0.6.5 with parameters $a = 6$ and $b = 8$. Interestingly, the intermediate integer value 7 between $a = 6$ and $b = 8$ can be excluded by means of a parity argument.

0.6.3 How to show that there is no PTAS: APX-hardness

Classical polynomial time reductions (as defined in Appendix 0.8) are the right tool for carrying over NP-hardness from one decision problem to another: If problem X is hard and if problem X polynomially reduces to another problem Y, then also this other problem Y is hard. Does there exist a similar tool for carrying over in-approximability from one optimization problem to another? Yes, *approximation preserving reductions* are such a tool. Loosely speaking, approximation preserving reductions are reductions that preserve approximability within a multiplicative constant. A lot of approximation preserving reductions have been introduced over the years, like the A-reduction, the F-reduction, the L-reduction, the P-reduction, the R-reduction, for which we refer the reader to the Ph.D. thesis of Kann [47]. In this chapter we will only deal with the L-reduction which is most practical for showing that one problem is as hard to approximate as another.

Definition 0.6.8 *(Papadimitriou & Yannakakis [64])*
Let X and Y be optimization problems. An L-reduction from problem X to problem Y is a pair of functions (R, S) that satisfy the following properties. R is a function from the set of instances of X to the set of instances of Y. S

is a function from the set of feasible solutions of the instances of Y to the set of feasible solutions of the instances of X. Both functions are computable in polynomial time.

(i) For any instance I of X with optimal cost $\text{OPT}(I)$, *$R(I)$ is an instance of Y with optimal cost* $\text{OPT}(R(I))$ *such that for some fixed positive constant α*

$$\text{OPT}(R(I)) \;\leq\; \alpha \cdot \text{OPT}(I).$$

(ii) For any feasible solution s of $R(I)$, $S(s)$ is a feasible solution of I such that for some fixed positive constant β

$$|c(S(s)) - \text{OPT}(I)| \;\leq\; \beta \cdot |c(s) - \text{OPT}(R(I))|.$$

Here $c(S(s))$ and $c(s)$ represent the costs of the feasible solutions $S(s)$ and s respectively.

Intuitively speaking, the two functions R and S draw a close connection between the approximability behavior of X and the approximability behavior of Y. By condition (ii), S is guaranteed to return a feasible solution of I which is not much more suboptimal than the given solution s of $R(I)$. Note also that S transforms optimal solutions of $R(I)$ into optimal solutions of I. And condition (i) simply states that the optimal values of I and $R(I)$ should stay relatively close together.

Theorem 0.6.9 *(Papadimitriou & Yannakakis [64])*
Assume that there exists an L-reduction with parameters α and β from the minimization problem X to the minimization problem Y. Assume furthermore that there exists a polynomial time approximation algorithm for Y with performance guarantee $1+\varepsilon$. Then there exists a polynomial time approximation algorithm for X with performance guarantee $1+\alpha\beta\varepsilon$.

Here is the straightforward proof of this theorem. Consider the following polynomial time approximation algorithm for X. For a given instance I of X, the algorithm (1) first computes the instance $R(I)$ of Y. (2) Then it applies the polynomial time approximation algorithm for Y with performance guarantee $1+\varepsilon$ to instance $R(I)$, and thus gets an approximate solution s. (3) Finally it computes the feasible solution $S(s)$ of I and outputs it as approximate solution for I. By Definition 0.6.8 all three steps can be done in polynomial time. Moreover, the cost $c(S(s))$ of the found approximate solution for I fulfills the following inequalities.

$$\begin{aligned} \frac{|c(S(s)) - \text{OPT}(I)|}{\text{OPT}(I)} &\leq \beta \, \frac{|c(s) - \text{OPT}(R(I))|}{\text{OPT}(I)} \\ &\leq \alpha\beta \, \frac{|c(s) - \text{OPT}(R(I))|}{\text{OPT}(R(I))} \quad \leq \quad \alpha\beta\varepsilon. \end{aligned}$$

For the first inequality we used condition (ii) from Definition 0.6.8, and for the second inequality we used condition (i) from Definition 0.6.8. The final inequality follows since we used a $(1+\varepsilon)$-approximation algorithm in the second step. This completes the proof.

Theorem 0.6.9 also holds true, if X is a maximization problem, or if Y is a maximization problem, or if both are maximization problems; see Exercise 0.6.9 in Section 0.6.5. Since $1+\alpha\beta\varepsilon$ comes arbitrarily close to 1 when $1+\varepsilon$ comes arbitrarily close to 1, we get the following immediate consequence.

Theorem 0.6.10 *Assume that X and Y are optimization problems, and that there is an L-reduction from X to Y. If there is a PTAS for Y, then there is a PTAS for X. Equivalently, if X does not have a PTAS, then Y does not have a PTAS.*

Sloppily speaking, L-reductions maintain the non-existence of PTAS's in very much the same way that classical polynomial time reductions maintain the non-existence of polynomial time algorithms. Next, we introduce the complexity class APX that plays the same role for L-reductions as NP plays for polynomial time reductions: The complexity class APX consists of all minimization and maximization problems that have a polynomial time approximation algorithm with some finite worst case ratio. An optimization problem is called APX-hard if every problem in APX can be L-reduced to it.

Many prominent optimization problems have been shown to be APX-hard, such as the maximum satisfiability problem, the maximum 3-dimensional matching problem, the problem of finding the maximum cut in a graph, the traveling salesman problem with the triangle-inequality. And for not a single one of these APX-hard optimization problems, a PTAS has been found! Theorem 0.6.10 yields that if there does exist a PTAS for *one* APX-hard problem, then *all* problems in APX have a PTAS. Moreover, in a breakthrough result from 1992 Arora, Lund, Motwani, Sudan & Szegedy [6] managed to establish a connection between APX-hardness and the infamous P=NP problem.

Theorem 0.6.11 *(Arora, Lund, Motwani, Sudan & Szegedy [6])*
If there exists a PTAS for some APX*-hard problem, then P=NP.*

So finally, here is the tool for proving that an optimization problem X cannot have a PTAS unless P=NP: Provide an L-reduction from an APX-hard problem to X. A detailed example of an L-reduction will be presented in Section 0.6.4. For more examples, we refer the reader to the Ph.D. thesis of Kann [47], or to the web-compendium of Crescenzi & Kann [17], or to the survey chapter of Arora & Lund [7]. We conclude this section by listing some in-approximability results in scheduling that have been derived via APX-hardness proofs.

- Hoogeveen, Schuurman & Woeginger [40] establish APX-hardness of several scheduling problems with the objective of minimizing the sum of all job completion times: The problems $R\,|\,r_j\,|\sum C_j$ and $R\,|\,|\sum w_jC_j$ on unrelated machines, and the shop problems $O\,|\,|\sum C_j$, $F\,|\,|\sum C_j$, and $J\,|\,|\sum C_j$ all are APX-hard.

- Hoogeveen, Skutella & Woeginger [41] show that preemptive scheduling with rejection on unrelated machines is APX-hard.

- Sviridenko & Woeginger [76] show that makespan minimization in no-wait job shops ($Jm \,|\, \textit{no-wait} \,|\, C_{\max}$) is APX-hard.

0.6.4 An example for an L-reduction

In this section we describe and discuss in detail an L-reduction (cf. Definition 0.6.8) from an APX-hard optimization problem X to an optimization problem Y. This L-reduction is taken from Azar, Epstein, Richter & Woeginger [13]. The optimization problem Y is the scheduling problem defined below in Example 0.6.12. The APX-hard optimization problem X considered is Maximum Bounded 3-Dimensional Matching (MAX-3DM-B).

The problem MAX-3DM-B is an optimization version of the 3-Dimensional Matching problem defined in Section 0.6.2. An instance of MAX-3DM-B consists of three sets $A = \{a_1, \ldots, a_q\}$, $B = \{b_1, \ldots, b_q\}$, and $C = \{c_1, \ldots, c_q\}$, together with a subset T of $A \times B \times C$. Any element in A, B, C occurs in one, two, or three triples in T; note that this implies $q \le |T| \le 3q$. The goal is to find a subset T' of T of maximum cardinality such that no two triples of T' agree in any coordinate. Such a set of triples is called *independent*. The measure of a feasible solution T' is the cardinality of T'. Petrank [66] has shown that MAX-3DM-B is APX-hard even if one only allows instances where the optimal solution consists of $q = |A| = |B| = |C|$ triples; in the following we will only consider this additionally restricted version of MAX-3DM-B.

Example 0.6.12 *An instance I of $R \,||\, \sum L_i^2$ is specified by m unrelated machines and n jobs. The processing time of job J_j ($j = 1, \ldots, n$) on machine M_i ($i = 1, \ldots, m$) is p_{ji}. In some fixed schedule we denote the total processing time (or load) assigned to machine M_i by L_i. The goal is to find a schedule that minimizes $\sum_{i=1}^m L_i^2$, the sum of squared machine loads.*

The problem $R \,||\, \sum L_i^2$ is strongly NP-hard. Awerbuch, Azar, Grove, Kao, Krishnan & Vitter [8] describe a polynomial time approximation algorithm with worst case ratio $3 + \sqrt{8}$ for $R \,||\, \sum L_i^2$. The problem $R \,||\, \sum L_i^2$ is APX-hard, since there exists an L-reduction from MAX-3DM-B *to $R \,||\, \sum L_i^2$. Consequently, $R \,||\, \sum L_i^2$ does not have a PTAS unless P=NP.*

For the L-reduction from MAX-3DM-B to $R \,||\, \sum L_i^2$ we first need to specify two functions R and S as required in Definition 0.6.8. Given any instance I of MAX-3DM-B, we construct an instance $R(I)$ of the scheduling problem $R \,||\, \sum L_i^2$. The instance $R(I)$ contains $3q$ machines. For every triple T_i in T, there is a corresponding machine $M(T_i)$. Moreover, there are $3q - |T|$ so-called dummy machines. The instance $R(I)$ contains $5q$ jobs. For every a_j, b_j, and c_j ($j = 1, \ldots, q$) there are corresponding element jobs $J(a_j)$, $J(b_j)$, and $J(c_j)$. Moreover there are $2q$ so-called dummy jobs. Now let us define the processing times of the jobs. Consider a triple $T_i = (a_j, b_k, c_l)$ and its corresponding machine $M(T_i)$. On machine $M(T_i)$, the processing time of the three element jobs $J(a_j)$, $J(b_k)$, and $J(c_l)$ equals 1. All other element jobs have infinite processing time on $M(T_i)$. All dummy jobs have processing time 3 on machine $M(T_i)$. On a dummy machine, all dummy jobs have processing time 3 and all element jobs have infinite processing time. This completes the description of the scheduling instance $R(I)$. Note that this construction is very similar to the polynomial time reduction that we used in the proof of Example 0.6.7 in Section 0.6.2.

Next we specify a function S as required in Definition 0.6.8. Let s be a feasible schedule for an instance $R(I)$ of $R\,||\sum L_i^2$. A machine $M(T_i)$ in the schedule s is called *good*, if it processes three jobs of length 1. Note that these three jobs can only be the jobs $J(a_j)$, $J(b_k)$, and $J(c_l)$ with $T_i = (a_j, b_k, c_l)$. We define the feasible solution $S(s)$ for the instance I of MAX-3DM-B to consist of all triples T_i for which the machine $M(T_i)$ is good.

Clearly, the two defined functions R and S both are computable in polynomial time. It remains to verify that they satisfy the conditions (i) and (ii) in Definition 0.6.8. Let us start by discussing condition (i). Since we only consider instances of MAX-3DM-B where the optimal solution consists of q triples, we have $\textsc{Opt}(I) = q$. Now consider the following schedule for instance $R(I)$: For each triple $T_i = (a_j, b_k, c_l)$ in the optimal solution to I, we schedule the three element jobs $J(a_j)$, $J(b_k)$, and $J(c_l)$ on machine $M(T_i)$. The $2q$ dummy jobs are assigned to the remaining $2q$ empty machines so that each machine receives exactly one dummy job. In the resulting schedule every machine has load 3, and hence the objective value of this schedule is $27q$. Consequently, $\textsc{Opt}(R(I)) \le 27q = 27\,\textsc{Opt}(I)$ and condition (i) is fulfilled with $\alpha = 27$.

Condition (ii) is more tedious to verify. Consider a feasible schedule s for an instance $R(I)$ of $R\,||\sum L_i^2$. Without loss of generality we may assume that in s every job has finite processing time; otherwise, the objective value $c(s)$ is infinite and condition (ii) is trivially satisfied. For $k = 0, 1, 2, 3$ let m_k denote the number of machines in schedule s that process exactly k jobs of length 1. Then the total number of machines equals

$$m_0 + m_1 + m_2 + m_3 \;=\; 3q, \tag{0.18}$$

and the total number of processed element jobs of length 1 equals

$$m_1 + 2m_2 + 3m_3 \;=\; 3q. \tag{0.19}$$

Note that by our definition of the function S, the objective value $c(S(s))$ of the feasible solution $S(s)$ equals m_3. Below we will prove that $c(s) \ge 29q - 2m_3$ holds. Altogether, this then will yield that

$$|c(S(s)) - \textsc{Opt}(I)| \;=\; q - m_3 \;=\; \frac{1}{2}(29q - 2m_3 - 27q) \;\le\; \frac{1}{2}|c(s) - \textsc{Opt}(R(I))|$$

and that condition (ii) is fulfilled with $\beta = 1/2$. Hence, it remains to prove $c(s) \ge 29q - 2m_3$. Let us remove all dummy jobs from schedule s and then add them again in the cheapest possible way, such that the resulting new schedule s' has the smallest possible objective value that can be reached by this procedure. Since $c(s) \ge c(s')$, it will be sufficient to establish the inequality $c(s') \ge 29q - 2m_3$. So, what is the cheapest way of adding the $2q$ dummy jobs of length 3 to m_0 empty machines, to m_1 machines with load 1, to m_2 machines with load 2, and to m_3 machines with load 3? It is quite straightforward to see that each machine should receive at most one dummy job, and that the dummy jobs should be added to the machines with the smallest loads. The inequality (0.19) implies $m_3 \le q$, and then (0.18) yields $m_0 + m_1 + m_2 \ge 2q$. Hence, the m_3 machines of load 3 will not receive any dummy job. The inequality (0.19) implies $m_1 + m_2 + m_3 \ge q$, and then (0.18) yields $m_0 \le 2q$. Hence, the m_0 empty machines all will receive a dummy job. For the rest of the argument we will distinguish two cases.

In the first case we assume that $m_0 + m_1 \ge 2q$. In this case there is sufficient space to accommodate all dummy jobs on the machines with load at most 1. Then schedule s' will have $m_0 + m_3$ machines of load 3, m_2 machines of load 2, $m_0 + m_1 - 2q$ machines of load 1, and $2q - m_0$ machines of load 4. From (0.18) and (0.19) we get that $m_0 = m_2 + 2m_3$ and that $m_1 = 3q - 2m_2 - 3m_3$. Moreover, our assumption $m_0 + m_1 \ge 2q$ is equivalent to $m_2 + m_3 - q \le 0$. We conclude that

$$\begin{aligned} c(s') &\ge 9(m_2 + 3m_3) + 4m_2 + (q - m_2 - m_3) + 16(2q - m_2 - 2m_3) \\ &= 33q - 4m_2 - 6m_3 \\ &\ge 33q - 4m_2 - 6m_3 + 4(m_2 + m_3 - q) \quad = \quad 29q - 2m_3. \end{aligned}$$

This completes the analysis of the first case. In the second case we assume that $m_0 + m_1 < 2q$. In this case there is not sufficient space to accommodate all dummy jobs on the machines with load at most 1, and some machines with load 2 must be used. Then schedule s' will have $m_0 + m_3$ machines of load 3, m_1 machines of load 4, $2q - m_0 - m_1$ machines of load 5, and $m_0 + m_1 + m_2 - 2q$ machines of load 2. As in the first case we use $m_0 = m_2 + 2m_3$ and $m_1 = 3q - 2m_2 - 3m_3$. Our assumption $m_0 + m_1 < 2q$ is equivalent to $q - m_2 - m_3 < 0$. We conclude that

$$\begin{aligned} c(s') &\ge 9(m_2 + 3m_3) + 16(3q - 2m_2 - 3m_3) + 25(m_2 + m_3 - q) + 4(q - m_3) \\ &= 27q + 2m_2 \;>\; 27q + 2m_2 + 2(q - m_2 - m_3) \;=\; 29q - 2m_3. \end{aligned}$$

This completes the analysis of the second case. To summarize, we have shown that also condition (ii) in Definition 0.6.8 is satisfied. We have established all the necessary properties of an L-reduction from MAX-3DM-B to $R\,||\,\sum L_i^2$. Hence, problem $R\,||\,\sum L_i^2$ indeed is APX-hard.

0.6.5 Exercises

Exercise 0.6.1. This exercise concerns the two Theorems 0.6.2 and 0.6.4 that were formulated in Section 0.6.1.

(a) Prove Theorem 0.6.2 by mimicking the arguments that we used in the discussion of Example 0.6.1. [Hint: Suppose that there is an FPTAS, and call it with some ε that satisfies $1/\varepsilon > p(|I|_{\text{unary}})$.]

(b) Prove Theorem 0.6.4 by mimicking the arguments that we used in the discussion of Example 0.6.3. [Hint: Suppose that there is an FPTAS, and call it with some ε that satisfies $1/\varepsilon > p(|I|)$.]

(c) Deduce Theorem 0.6.2 from Theorem 0.6.4. [Hint: Read the chapter in Garey & Johnson [24] on strong NP-hardness. For a problem X and a polynomial q Garey & Johnson define the subproblem X_q of X that consists of all instances I of X in which the value of the largest occuring number is bounded by $q(|I|)$. Garey & Johnson say that a problem X is strongly NP-hard, if there exists a polynomial q such that the subproblem X_q is NP-hard. Now consider a strongly NP-hard optimization problem X that satisfies the conditions of Theorem 0.6.2. Argue that for some appropriately defined polynomial q, the corresponding subproblem X_q satisfies the conditions of Theorem 0.6.4.]

Exercise 0.6.2. Decide whether the following strongly NP-hard minimization problems have an FPTAS. Note that you cannot apply Theorem 0.6.2 to these problems! As in $P\,||\,C_{\max}$, the input consists of n positive integer processing times p_j $(j = 1, \ldots, n)$ and of m identical machines. The objective functions are the following.

(a) For a schedule σ with maximum machine load $C_{\max}$ and minimum machine load $C_{\min}$, the objective value is $f(\sigma) = C_{\max}/C_{\min}$. [Hint: Use the fact that even the following decision version of $P\,||\,C_{\max}$ is strongly NP-hard. Given an instance, does there exist a feasible schedule in which all machines have equal load.]

(b) For a schedule σ with makespan $C_{\max}$, the objective value is $f(\sigma) = 1 - 1/C_{\max}$.

(c) For a schedule σ with makespan $C_{\max}$, the objective value is $f(\sigma) = 4 - 1/2^{C_{\max}}$. [Hint: If ε is large and satisfies $\log(1/\varepsilon) \le p_{\text{sum}}$, then you may come within a factor of $1 + \varepsilon$ of optimal by using the algorithm that is described in the paragraph after inequality (0.6) in Section 0.3.2. If ε is small and satisfies $\log(1/\varepsilon) \ge p_{\text{sum}}$, then you have a lot of time for solving the instance by some slower approach.]

Exercise 0.6.3. Consider the following minimization problem X. As in $P2\,||\,C_{\max}$, the input consists of n positive integer processing times p_j $(j = 1, \ldots, n)$. The goal is to find a schedule σ on two identical machines that minimizes the following objective function $f(\sigma)$: If the makespan of σ is at most $\frac{1}{2}\sum_{j=1}^{n} p_j$ then $f(\sigma) = 8$, and otherwise $f(\sigma) = 11$.

(a) Show that problem X is solvable in pseudo-polynomial time.

(b) Show that problem X cannot have an FPTAS unless P=NP. Can you get a PTAS for X? What is the best worst case guarantee that you can reach for a polynomial time approximation algorithm for X?

(c) Construct a minimization problem Y that satisfies the following two properties: Y is pseudo-polynomially solvable. Unless P=NP, there is no polynomial time approximation algorithm with finite worst case guarantee for Y.

Exercise 0.6.4. Consider two knapsacks of size b and n items, where the jth item ($j = 1, \ldots, n$) has a positive integer size a_j. The goal is to pack the maximum *number* of items into the two knapsacks. Prove that the existence of an FPTAS for this problem would imply P=NP.

Exercise 0.6.5. Reformulate and prove the Theorems 0.6.5 and 0.6.6 for maximization problems.

Exercise 0.6.6. Consider n jobs with processing times p_j ($j = 1, \ldots, n$) and a hard deadline d. The goal is to find the minimum number m of machines on which all jobs can be completed before their deadline d.

Show that for this problem deciding whether $\text{OPT} \leq 2$ is NP-hard. Use Lenstra's impossibility theorem (Theorem 0.6.6) to deduce an in-approximability result from this.

Exercise 0.6.7. An instance of the problem $P \,|\, prec, p_j{=}1 \,|\, C_{\max}$ consists of n precedence constrained unit-time jobs and of m identical machines. The goal is to find a schedule that minimizes the makespan while obeying the precedence constraints (all predecessor jobs must be completed before a successor job can start). The goal in the closely related problem $P \,|\, prec, p_j{=}1 \,|\, \sum C_j$ is to find a schedule that minimizes the sum of all the job completion times.

Lenstra & Rinnooy Kan [58] proved for $P \,|\, prec, p_j{=}1 \,|\, C_{\max}$ that deciding whether $\text{OPT} \leq 3$ is NP-hard: They provide a polynomial time transformation from the NP-hard clique problem in graphs (the only thing you need to know about the clique problem is that it is NP-hard). Given an instance of clique, this transformation computes a set of m machines and $3m$ precedence constrained jobs with unit processing times. In the case of a YES-instance, the constructed scheduling instance has a schedule where all $3m$ jobs are processed during the time interval $[0, 3]$. In the case of a NO-instance, in every feasible schedule for the constructed scheduling instance at least one job completes at time 4 or later.

(a) Argue that unless P=NP, the problem $P \,|\, prec, p_j{=}1 \,|\, C_{\max}$ cannot have a polynomial time approximation algorithm with worst case ratio $4/3 - \varepsilon$. [Hint: Apply Theorem 0.6.6.]

(b) Is it difficult to decide whether $\text{OPT} \leq 2$ holds for an instance of $P \,|\, prec, p_j{=}1 \,|\, C_{\max}$?

(c) Prove that unless P=NP, the problem $P \,|\, prec, p_j{=}1 \,|\, \sum C_j$ cannot have a polynomial time approximation algorithm with worst case ratio $4/3 - \varepsilon$. [Hint: Create many copies of the Lenstra & Rinnooy Kan instance and put them in series.]

Exercise 0.6.8. Consider an instance of the open shop problem $O\,||\,C_{\max}$ with an arbitrary number of machines where all operations have integer lengths. In an open shop every job J_j must be run on every machine M_i for p_{ij} time units. An operation is just a maximal piece of J_j that is processed on the same machine. In the non-preemptive version every operation must be processed on its machine without any interruptions; in the preemptive version operations may be interrupted and resumed later on. Computing the optimal non-preemptive makespan is NP-hard, whereas computing the optimal preemptive makespan is easy. It is known (and easy to show) that the optimal preemptive makespan equals the maximum of the maximum machine load and the maximum job length.

Suppose, you are given an instance I of $O\,||\,C_{\max}$ whose preemptive makespan is four. Does there always exist a non-preemptive schedule for I with makespan at most (a) seven, (b) six, (c) five, (d) four? And in case such a non-preemptive schedule does not always exist, can you at least decide its existence in polynomial time? [Remark: If you are able to settle question (c) completely, then please send us your argument.]

Exercise 0.6.9. We only stated Theorem 0.6.9 for the case where X and Y both are minimization problems. Check the other three cases where X might be a maximization problem (in this case the new conclusion should be that problem X has a polynomial time approximation algorithm with worst case ratio $1-\alpha\beta\varepsilon$) and where Y might be a maximization problem (in this case the new assumption should be that Y has a polynomial time approximation algorithm with worst case ratio $1-\varepsilon$).

Exercise 0.6.10. Let $p > 1$ be a real number. Consider the scheduling problem $R\,||\,\sum L_i^p$, i.e., the problem of minimizing the sum of the p-th powers of the machine loads on unrelated machines.

(a) Prove that $R\,||\,\sum L_i^p$ is APX-hard for $p = 3$. [Hint: Modify the argument in Section 0.6.4.]

(b) Prove that $R\,||\,\sum L_i^p$ is APX-hard for all values $p > 1$.

Exercise 0.6.11. Consider n jobs J_j ($j = 1, \ldots, n$) with positive integer processing times p_j on two identical machines. The goal is to find a schedule that minimizes $(L_1 - L_2)^2$ where L_1 and L_2 are the machine loads.

Decide on the existence of (a) a PTAS, (b) an FPTAS, (c) a polynomial time approximation algorithm with finite worst case ratio for this problem under the assumption P$\neq$NP.

Exercise 0.6.12. Consider an NP-hard scheduling problem X. Denote by I an instance of this problem, by σ a feasible schedule for I, and by $f(\sigma)$ the objective value of schedule σ. Assume that it is NP-hard to determine for instance I a schedule σ that minimizes the value $f(\sigma)$. Consider the following related problem X'. An instance of X' consists of an instance I of X together with a positive integer T. The goal is to find a schedule σ for I such that $(T - f(\sigma))^2$ is minimized.

Decide on the existence of (a) a PTAS, (b) an FPTAS, (c) a polynomial time approximation algorithm with finite worst case ratio for the minimization problem X' under the assumption P≠NP.

Exercise 0.6.13. Paz & Moran [65] call a minimization problem X *simple*, if for every fixed positive integer k the problem of deciding whether an instance I of X has optimal objective value at most k is polynomially solvable.

(a) Prove that the problems $P2\,||\,C_{\max}$ and $R2\,||\,C_{\max}$ are simple.

(b) Decide which of the problems $P\,||\,C_{\max}$, $R\,||\,C_{\max}$, and $1\,||\,\sum T_j$ (see Section 0.3.3) are simple under the assumption P≠NP.

(c) Prove: If the minimization problem X has a PTAS, then it is simple.

(d) Find a minimization problem Y that is simple, but does not have a PTAS unless P=NP. Can you establish the in-approximability of Y via the gap technique?

0.7 Conclusions and further reading

In this chapter we have discussed many approximation schemes together with several in-approximability proofs. We tried to identify the main mechanisms and to illuminate the underlying machinery. However, we did not try at all to present very short or very polished arguments; our main goal was to communicate the ideas and to make them comprehensible to the reader. For instance, for makespan minimization on two identical machines we described three independent approximation schemes that illustrate the three main conceptual approaches to approximation schemes. And our presentation of Lawler's FPTAS for total tardiness on a single machine in Section 0.3.3 takes a lot more space than the presentation of the same result in the original paper [54]. We also stress that our illustrative examples describe clean *prototypes* for the three main approaches to polynomial time approximation schemes, whereas real life approximation schemes usually mix and intertwine the three approaches.

Although this chapter has explained quite a few tricks for designing approximation schemes for scheduling problems, there are still more tricks to learn. Fernandez de la Vega & Lueker [20] introduce a nice data simplification technique for bin packing. The big numbers are partitioned into lots of groups of equal cardinality, and then all elements in a group are replaced by the maximum or minimum element in the group. Sevastianov & Woeginger [74] classify jobs into big, medium, and small jobs (instead of using only big and small jobs, as we have been doing all the time). One can get rid of the medium jobs, and then the small jobs are clearly separated by a gap from the big jobs. Amoura, Bampis, Kenyon & Manoussakis [4] use linear programming in a novel way to get a PTAS for scheduling multiprocessor tasks. The breakthrough paper by Afrati et al. [2] contains a collection of approximation schemes for all kinds of scheduling problems with the objective of minimizing the sum of job completion times. It is instructive to read carefully through this paper, and to understand every single rounding, classification, or simplification step (these steps are linked and woven into each other).

Further reading. During the last few years, three excellent books on approximation algorithms have been published: The book edited by Hochbaum [35] contains thirteen chapters that are written by the experts in the field and that cover all facets of approximation algorithms. The book by Ausiello, Crescenzi, Gambiosi, Kann, Marchetti-Spaccamela & Protasi [9] looks at approximability and in-approximability from the viewpoint of computational complexity theory. The book by Vazirani [78] concentrates on algorithms and discusses all the pearls in the area. To learn more about in-approximability techniques, we recommend the Ph.D. thesis of Kann [47], the web-compendium by Crescenzi & Kann [17], the survey chapter by Arora & Lund [7], and the paper by Lund & Yannakakis [62].

Now let us turn to specialized literature on scheduling. The survey by Hall [30] provides a nice introduction into approximation algorithms for scheduling; it discusses simple greedy algorithms, approximation schemes, linear programming relaxations, probabilistic arguments, etc etc. The survey by Lenstra & Shmoys [59] mainly deals with in-approximability results that can be derived via the gap technique. Schuurman & Woeginger [72] list the top open problems in the approximation of scheduling; they summarize what is known on these problems, discuss related results, and provide pointers to the literature.

Acknowledgements. We thank Eranda Çela, Bettina Klinz, John Noga, and Laurence Wolsey for proof-reading preliminary versions of this chapter and for many helpful comments. Gerhard Woeginger acknowledges support by the START program Y43-MAT of the Austrian Ministry of Science.

0.8 Computational complexity

Computational complexity theory provides a classification of computational problems into easy and hard. We briefly sketch some of the main points of this theory. For more information, the reader is referred to the books by Garey & Johnson [24] and Papadimitriou [63].

A computational problem consists of a set $\mathcal{I}$ of inputs (or instances) together with a function that assigns to every input $I \in \mathcal{I}$ a corresponding output (or solution). The input size (or length) $|I|$ of an input I is the overall number of bits used for encoding I under some given encoding scheme. The time complexity of an algorithm for a computational problem is measured as a function of the size of the input. We say that a computational problem is *easy* to solve if there is a polynomial time algorithm for it, i.e., if there exists a polynomial p such that the algorithm applied to an input I always finds a solution in time at most $p(|I|)$. The complexity class P is the set of all easy problems.

The concept of hardness of a computational problem is formalized with respect to the complexity class NP. The class NP only contains decision problems, i.e., computational problems for which the outputs are restricted to the set {YES, NO}. Since by means of binary search one can represent every optimization problem as a short sequence of decision problems, this restriction is not really essential; all of the following statements and terms easily carry over to optimization problems as well (and actually, in the rest of this chapter we are going to use these terms mainly for optimization problems). Loosely speaking, the class NP contains all decision problems for which each YES-instance I has a certificate $c(I)$ such that $|c(I)|$ is bounded by a polynomial in $|I|$ and such that one can verify in polynomial time that $c(I)$ is a valid certificate for I. For a precise definition of the class NP we refer the reader to Garey & Johnson [24]. In order to compare the computational complexities of two decision problems X and Y, the concept of a reduction is used: We say that problem X reduces to problem Y, if there exists a tranformation that maps any YES-instance of X into a YES-instance of Y and any NO-instance of X into a NO-instance of Y. If the reduction can be performed in polynomial time, then a polynomial time algorithm for problem Y will automatically yield a polynomial time algorithm for problem X. Intuitively, if X reduces to Y then Y is at least as hard as X. A decision problem is called NP-hard if every problem in NP can be polynomially reduced to it. Unless P=NP, which is considered extremely unlikely, an NP-hard problem does not possess a polynomial algorithm. And if a *single* NP-hard problem allowed a polynomial algorithm, then *all* problems in NP would allow polynomial algorithms. The gist of this is that NP-hard problems are considered to be hard and intractable.

There are two basic schemes of encoding the numbers in an input: One scheme is the standard binary encoding that is used in every standard computer. Another scheme is the unary encoding where any integer k is encoded by k bits. Note that the concepts of polynomial solvability and NP-hardness crucially depend on the encoding scheme used. If one changes the encoding scheme from binary to unary, the problem may become easier, as the input becomes longer and hence the restrictions on the time complexity of a polynomial algorithm are less stringent. A problem that is NP-hard under the unary encoding scheme is called strongly NP-hard. A problem that can be solved in polynomial time under the unary encoding scheme is said to be pseudo-polynomially solvable. The corresponding algorithm is called a pseudo-polynomial algorithm. Pseudo-polynomially solvable problems that are NP-hard under binary encodings are sometimes said to be NP-hard in the ordinary sense. If a *single* strongly NP-hard problem allowed a pseudo-polynomial algorithm, then *all* problems in NP would allow polynomial algorithms and P=NP would hold.

Bibliography

[1] S. Adams [1996]. The Dilbert principle: A cubicle's-eye view of bosses, meetings, management fads & other workplace afflictions. HarperCollins Publishers.

[2] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko [1999]. Approximation schemes for minimizing average weighted completion time with release dates. *Proceedings of the 40th IEEE Symposium on the Foundations of Computer Science (FOCS'99)*, 32–43.

[3] N. Alon, Y. Azar, G.J. Woeginger, and T. Yadid [1998]. Approximation schemes for scheduling on parallel machines. *Journal of Scheduling 1*, 55–66.

[4] A.K. Amoura, E. Bampis, C. Kenyon, and Y. Manoussakis [1997]. Scheduling independent multiprocessor tasks. *Proceedings of the 5th European Symposium on Algorithms (ESA'97)*, Springer LNCS 1284, 1–12.

[5] S. Arora [1996]. Polynomial-time approximation schemes for Euclidean TSP and other geometric problems. *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS'96)*, 2–11.

[6] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy [1992]. Proof verification and hardness of approximation problems. *Proceedings of the 33rd Annual IEEE Symposium on the Foundations of Computer Science (FOCS'92)*, 14–23.

[7] S. Arora and C. Lund [1997]. Hardness of approximation. In: D.S. Hochbaum (ed.) *Approximation algorithms for NP-hard problems.* PWS Publishing Company, Boston, 1–45.

[8] B. Awerbuch, Y. Azar, E. Grove, M. Kao, P. Krishnan, and J. Vitter [1995]. Load balancing in the L_p norm. *Proceedings of the 36th Annual IEEE Symposium on the Foundations of Computer Science (FOCS'95)*, 383–391.

[9] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi [1999]. *Complexity and Approximation.* Springer, Berlin.

[10] G. Ausiello, P. Crescenzi, and M. Protasi [1995]. Approximate solution of NP optimization problems. *Theoretical Computer Science 150*, 1–55.

[11] G. Ausiello, A. D'Atri, and M. Protasi [1980]. Structure preserving reductions among convex optimization problems. *Journal of Computer and Systems Sciences 21*, 136–153.

[12] G. Ausiello, A. Marchetti-Spaccamela, and M. Protasi [1980]. Toward a unified approach for the classification of NP-complete optimization problems. *Theoretical Computer Science 12*, 83–96.

[13] Y. Azar, L. Epstein, Y. Richter, and G.J. Woeginger [2001]. All-norm approximation algorithms. Manuscript.

[14] B.S. Baker [1994]. Approximation algorithms for NP-complete problems on planar graps. *Journal of the ACM 41*, 153–180.

[15] R.E. Bellman and S.E. Dreyfus [1962]. *Applied Dynamic Programming.* Princeton University Press.

[16] J.L. Bruno, E.G. Coffman, Jr., and R. Sethi [1974]. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM 17*, 382–387.

[17] P. Crescenzi and V. Kann. A compendium of NP-optimization problems. http:// www.nada.kth.se/nada/theory/problemlist.html.

[18] J. Du and J.Y.T. Leung [1990]. Minimizing total tardiness on one machine is NP-hard. *Mathematics of Operations Research 15*, 483–495.

[19] D.W. Engels, D.R. Karger, S.G. Kolliopoulos, S. Sengupta, R.N. Uma, and J. Wein [1998]. Techniques for scheduling with rejection. *Proceedings of the 6th European Symposium on Algorithms (ESA'98)*, Springer LNCS 1461, 490–501.

[20] W. Fernandez de la Vega and G.S. Lueker [1981]. Bin packing can be solved within $1+\varepsilon$ in linear time. *Combinatorica 1*, 349–355.

[21] M.R. Garey, R.L. Graham, and J.D. Ullman [1972]. Worst case analysis of memory allocation algorithms. *Proceedings of the 4th Annual ACM Symposium on the Theory of Computing (STOC'72)*, 143–150.

[22] M.R. Garey and D.S. Johnson [1976]. The complexity of near-optimal graph coloring. *Journal of the ACM 23*, 43–49.

[23] M.R. Garey and D.S. Johnson [1978]. 'Strong' NP-completeness results: Motivation, examples, and implications. *Journal of the ACM 25*, 499–508.

[24] M.R. Garey and D.S. Johnson [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco.

[25] G.V. GENS AND E.V. LEVNER [1981]. Fast approximation algorithms for job sequencing with deadlines. *Discrete Applied Mathematics 3*, 313–318.

[26] R.L. GRAHAM [1966]. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal 45*, 1563–1581.

[27] R.L. GRAHAM [1969]. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics 17*, 416–429.

[28] R.L. GRAHAM, E.L. LAWLER, J.K. LENSTRA, AND A.H.G. RINNOOY KAN [1979]. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics 5*, 287–326.

[29] L.A. HALL [1994]. A polynomial time approximation scheme for a constrained flow shop scheduling problem. *Mathematics of Operations Research 19*, 68–85.

[30] L.A. HALL [1997]. Approximation algorithms for scheduling. In: D.S. Hochbaum (ed.) *Approximation algorithms for NP-hard problems.* PWS Publishing Company, Boston, 1–45.

[31] L.A. HALL [1998]. Approximability of flow shop scheduling. *Mathematical Programming 82*, 175–190.

[32] L.A. HALL AND D.B. SHMOYS [1989]. Approximation schemes for constrained scheduling problems. *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS'89)*, 134–139.

[33] L.A. HALL AND D.B. SHMOYS [1990]. Near-optimal sequencing with precedence constraints. *Proceedings of the 1st Conference on Integer Programming and Combinatorial Optimization (IPCO'90)*, 249–260.

[34] L.A. HALL AND D.B. SHMOYS [1992]. Jackson's rule for single-machine scheduling: Making a good heuristic better. *Mathematics of Operations Research 17*, 22–35.

[35] D.S. HOCHBAUM (ED.) [1997]. *Approximation algorithms for NP-hard problems.* PWS Publishing Company, Boston.

[36] D.S. HOCHBAUM AND W. MAASS [1985]. Approximation schemes for covering and packing problems in image processing and VLSI. *Journal of the ACM 32*, 130–136.

[37] D.S. HOCHBAUM AND D.B. SHMOYS [1987]. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM 34*, 144–162.

[38] D.S. HOCHBAUM AND D.B. SHMOYS [1988]. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing 17*, 539–551.

[39] J.A. Hoogeveen, J.K. Lenstra, and B. Veltman [1994]. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters 16*, 129–137.

[40] J.A. Hoogeveen, P. Schuurman, and G.J. Woeginger [1998]. Non-approximability results for scheduling problems with minsum criteria. *Proceedings of the 6th Conference on Integer Programming and Combinatorial Optimization (IPCO'98)*, Springer LNCS 1412, 353–366.

[41] J.A. Hoogeveen, M. Skutella, and G.J. Woeginger [2000]. Preemptive scheduling with rejection. *Proceedings of the 8th European Symposium on Algorithms (ESA'2000)*, Springer LNCS 1879, 268–277.

[42] E. Horowitz and S. Sahni [1974]. Computing partitions with applications to the knapsack problem. *Journal of the ACM 21*, 277–292.

[43] E. Horowitz and S. Sahni [1976]. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM 23*, 317–327.

[44] O. Ibarra and C.E. Kim [1975]. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM 22*, 463–468.

[45] D.S. Johnson [1974]. Approximation algorithms for combinatorial problems. *Journal of Computer and Systems Sciences 9*, 256–278.

[46] D.S. Johnson [1992]. The tale of the second prover. *Journal of Algorithms 13*, 502–524.

[47] V. Kann [1991]. *On the approximability of NP-complete optimization problems.* Royal Institute of Technology, Stockholm, Sweden.

[48] R.M. Karp [1972]. Reducibility among combinatorial problems. In: R.E. Miller and J.W. Thatcher (eds.) *Complexity of Computer Computations*. Plenum Press, New York, 85–104.

[49] H. Kellerer, T. Tautenhahn, and G.J. Woeginger [1999]. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM Journal on Computing 28*, 1155–1166.

[50] B. Korte and R. Schrader [1981]. On the existence of fast approximation schemes. In: O.L. Mangasarian, R.R. Meyer, and S.M. Robinson (eds.) *Nonlinear Programming.* Academic Press, New York, 415–437.

[51] M.Y. Kovalyov and W. Kubiak [1998]. A fully polynomial time approximation scheme for minimizing makespan of deteriorating jobs. *Journal of Heuristics 3*, 287–297.

[52] M.Y. Kovalyov, C.N. Potts, and L.N. Van Wassenhove [1994]. A fully polynomial approximation scheme for scheduling a single machine to minimize total weighted late work. *Mathematics of Operations Research 19*, 86–93.

[53] E.L. LAWLER [1977]. A 'pseudopolynomial' algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics 1*, 331–342.

[54] E.L. LAWLER [1982]. A fully polynomial approximation scheme for the total tardiness problem. *Operations Research Letters 1*, 207–208.

[55] E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, AND D.B. SHMOYS [1993]. Sequencing and scheduling: Algorithms and complexity. In: S.C. Graves, A.H.G. Rinnooy Kan, and P.H. Zipkin (eds.) *Logistics of Production and Inventory*, Handbooks in Operations Research and Management Science 4. North-Holland, Amsterdam, 445–522.

[56] E.L. LAWLER AND J.M. MOORE [1969]. A functional equation and its application to resource allocation and sequencing problems. *Management Science 16*, 77–84.

[57] H.W. LENSTRA [1983]. Integer programming with a fixed number of variables. *Mathematics of Operations Research 8*, 538–548.

[58] J.K. LENSTRA AND A.H.G. RINNOOY KAN [1978]. Complexity of scheduling under precedence constraints. *Operations Research 26*, 22–35.

[59] J.K. LENSTRA AND D.B. SHMOYS [1995]. Computing near-optimal schedules. In: P. Chrétienne, E.G. Coffman, Jr., J.K. Lenstra, and Z. Liu (eds.) *Scheduling Theory and its Applications*. Wiley, Chichester, 1–14.

[60] J.K. LENSTRA, D.B. SHMOYS, AND É. TARDOS [1990]. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming 46*, 259–271.

[61] S. LEONARDI AND D. RAZ [1997]. Approximating total flow time on parallel machines. *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC'97)*, 110–119.

[62] C. LUND AND M. YANNAKAKIS [1994]. On the hardness of approximating minimization problems. *Journal of the ACM 41*, 960–981.

[63] C.H. PAPADIMITRIOU [1994]. *Computational Complexity*. Addison-Wesley.

[64] C.H. PAPADIMITRIOU AND M. YANNAKAKIS [1991]. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences 43*, 425–440.

[65] A. PAZ AND S. MORAN [1981]. Non deterministic polynomial optimization problems and their approximations. *Theoretical Computer Science 15*, 251–277.

[66] E. PETRANK [1994]. The hardness of approximation: Gap location. *Computational Complexity 4*, 133–157.

[67] C. POTTS [1985]. Analysis of a linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics 10*, 155-164.

[68] C.N. Potts and L.N. Van Wassenhove [1992]. Approximation algorithms for scheduling a single machine to minimize total late work. *Operations Research Letters 11*, 261–266.

[69] S. Sahni [1975]. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM 22*, 115–124.

[70] S. Sahni [1976]. Algorithms for scheduling independent tasks. *Journal of the ACM 23*, 116–127.

[71] S. Sahni and T.F. Gonzalez [1976]. P-complete approximation problems. *Journal of the ACM 23*, 555-565.

[72] P. Schuurman and G.J. Woeginger [1999]. Polynomial time approximation algorithms for machine scheduling: Ten open problems. *Journal of Scheduling 2*, 203–213.

[73] P. Schuurman and G.J. Woeginger [2000]. Scheduling a pipelined operator graph. *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'2000)*, 207–212.

[74] S.V. Sevastianov and G.J. Woeginger [1998]. Makespan minimization in open shops: A polynomial time approximation scheme. *Mathematical Programming 82*, 191–198.

[75] W.E. Smith [1956]. Various optimizers for single-stage production. *Naval Research Logistics Quarterly 3*, 59–66.

[76] M. Sviridenko and G.J. Woeginger [2000]. Approximability and inapproximability results for no-wait shop scheduling. *Proceedings of the 41st IEEE Symposium on the Foundations of Computer Science (FOCS'2000)*, 116–125.

[77] C.P.M. van Hoesel and A.P.M. Wagelmans [1997]. Fully polynomial approximation schemes for single-item capacitated economic lot-sizing problems. Economic Institute Report 9735/A, Erasmus University Rotterdam.

[78] V. Vazirani [2000]. *Approximation algorithms.* Springer, New-York.

[79] D.P. Williamson, L.A. Hall, J.A. Hoogeveen, C.A.J. Hurkens, J.K. Lenstra, S.V. Sevastianov, and D.B. Shmoys [1997]. Short shop schedules. *Operations Research 45*, 288–294.

[80] G.J. Woeginger [1999]. When does a dynamic programming formulation guarantee the existence of an FPTAS? *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, 820–829.