Sven O. Krumke

Online Optimization

OptALI Summer School Auckland, 2011

Draft: February 8, 2011

Contents

1 Basic Concepts

1.1 A Simple Problem?

A situation which many of us know: You overslept, you are already late for the morning meeting, all traffic lights are on red, and once you finally reach the office building it takes ages for the elevator to arrive. Who on earth designed this elevator control system? There must be a way to craft a perfect elevator control with a little bit of mathematics!

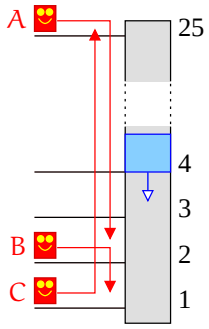

Figure 1.1: The initial situation: A, B, and C request transportations.

Let us consider the situation of an elevator that runs in a 25 floor building and which currently is waiting at the fourth floor. Amalia requests a transportation from the penthouse in the 25th floor down to the second floor, Beatrice wants to be moved from the second floor to the first, and Claus wants to go upwards from the first floor to the penthouse, see Figure 1.1.

Our current task is easy: we obtain a shortest transportation, if we first pick up Beatrice, then Claus, and finally take care of Amalia.

However, just a second after we pass the third floor on our way down, suddenly David appears at the third floor and wants to be carried to the second floor. Hmm..., what shall we do? Should we first complete our initial transportation schedule and care about David later? Or should we reverse direction, and pick up David first?

In any case we waste valuable time since we travel unnecessary distance which we could have avoided *if we had known in advance* that (and when) David showed up.

We have just discovered the *online aspect* of the elevator problem. We are facing incomplete information, and even if every time a new request becomes known we compute a

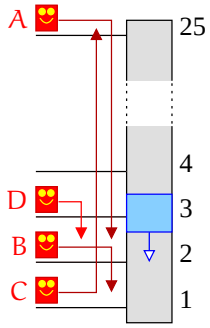

Figure 1.2: D appears ...

new "optimal" schedule this does not necessarily lead to an overall optimal solution. Suppose that in our particular case David were actually the last transportation request and our goal would be to finish serving requests as early as possible. Then, *in hindsight* (or *with clairvoyance*) the best solution would have been to *wait* at the third floor for one second until David arrived.

At the moment our most promising option looks like reversing gear and handling David first (after all, we have just lost one second right now compared to a clairvoyant controller). But what if Emma shows up on the fourth floor, just before we pick up David? Should we then serve Emma first?

Perhaps the elevator problem is not so trivial!

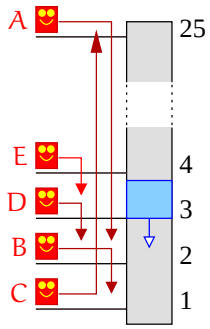

Figure 1.3: Is this the future?

1.2 Online Computation

In general, traditional optimization techniques assume complete knowledge of all data of a problem instance. However, in reality it is unlikely that all information necessary to define a problem instance is available beforehand. Decisions may have to be made before complete information is available. This observation has motivated the research on *online optimization*. An algorithm is called *online* if it makes a decision (computes a partial solution) whenever a new piece of data requests an action.

Let us illustrate *online computation* with a few more examples.

Example 1.1 (Ski Rental Problem)
Mabel goes skiing for the first time in her life. She is faced with the question of whether to buy skis for B € (B $\gg$ 1) or to rent skis at the cost of 1 € per day. Of course, if Mabel

knew how many times she would go skiing in the future, her decision would be easy. But unfortunately, she is in an online situation where the number of skiing days only becomes known at the very last day. ◁

Example 1.2 (Ice Cream Problem)
Luigi owns a small ice cream shop where he only sells the two most popular ice cream flavors, vanilla and chocolate in dishes of one liter. His ice cream machine has two modes, V and C for producing vanilla and chocolate ice cream, respectively. Switching between modes requires Luigi to clean the machine which amounts to costs of 1€. Upon the press of a button, the machine produces the sort of ice cream which corresponds to its current mode. The corresponding cost is given in the second column of Table 1.1.

Instead of using the machine, Luigi may also produce ice cream manually, which costs him a bit more, since it eats up a substantial amount of his time. The cost is given in the third column of Table 1.1.

Flavor	Machine Cost	Manual Cost
V	1€	2€
C	2€	4€

Table 1.1: Cost for producing ice cream by machine and manually. The cost for switching the mode at the machine is 1€

.

People queue in front of Luigi's cafe. Hence, Luigi can only see the first ice cream request in the line and must serve this request before learning the next order. Which requests should Luigi serve by using the machine and which ones should he serve manually in order to minimize his total cost? ◁

Our third problem concerns a more serious problem from the area of scheduling.

Example 1.3 (Online Machine Scheduling with Jobs Arriving Over Time)
In scheduling one is concerned with the assignment of jobs (activities) to a number of machines (the resources). In our example, one is given m identical machines and is faced with the task of scheduling independent jobs on these machines. The jobs become available at their release dates, specifying their processing times. An online algorithm learns the existence of a job only at its release date. Once a job has been started on a machine the job may not be preempted and has to run until completion. However, jobs that have been scheduled but not yet started may be rescheduled. The objective is to minimize the average flow time of a job, where the *flow time* of a job is defined to be the difference between the completion time of the job and its release date. ◁

All of our problems have in common that decisions have to be made without knowledge of the future. Notice that there is a subtle difference between the ski rental problem and the ice cream problem on the one hand and the elevator and scheduling problems on the other hand: In the the ski rental and ice cream problems a request demands an immediate answer which must be given before the next request is revealed. In the other two problems, an online algorithm is allowed to wait and to revoke decisions. Waiting incurs additional costs, typically depending on the elapsed time. Previously made decisions may, of course, only be revoked as long as they have not been executed.

These two different models are known as *sequence paradigm* and *time stamp paradigm* for online problems. In the *sequence paradigm* requests must be served in the order of their occurrence. More precisely, when serving request r_j, an online algorithm ALG does not have any knowledge of requests r_i with $i > j$ (or the total number of requests). When

request r_j is presented it must be served by ALG according to the specific rules of the problem. The serving of r_j incurs a "cost" and the overall goal is to minimize the total service cost.[1] The decision by ALG of how to serve r_j is irrevocable. Only after r_j has been served, the next request r_{j+1} becomes known to ALG.

In the *time stamp paradigm* each request has a *arrival* or *release time* at which it becomes available for service. The release time $t_j \geqslant 0$ is a nonnegative real number and specifies the time at which request r_j is released (becomes known to an online algorithm). An online algorithm ALG must determine its behavior at a certain moment t in time as a function of all the requests released up to time t and of the current time t. Again, we are in the situation that an online algorithm ALG is confronted with a finite sequence $r_1, r_2, \ldots$ of requests which is given in order of non-decreasing release times and the service of each request incurs a cost for ALG. The difference to the sequence paradigm is that the online algorithm is allowed to wait and to revoke decisions and that requests need not be served in the order of their occurrence. Waiting incurs additional costs, typically depending on the elapsed time. Previously made decisions may, of course, only be revoked as long as they have not been executed.

A general model that comprises many online problems is the concept of a request-answer game:

Definition 1.4 (Request-Answer Game)
*A **request-answer game** $(R, \mathcal{A}, \mathcal{C})$ consists of a request set R, a sequence of nonempty answer sets $\mathcal{A} = A_1, A_2, \ldots$ and a sequence of cost functions $\mathcal{C} = C_1, C_2, \ldots$ where $C_j \colon R^j \times A_1 \times \cdots \times A_j \to \mathbb{R}_+ \cup \{+\infty\}$.*

We remark here that in the literature one frequently requires each answer set A_j to be finite. This assumption is made to avoid difficulties with the existence of expected values when studying randomization. However, the finiteness requirement is not of conceptual importance. As remarked in [6, Chapter 7] an infinite or even continuous answer set can be "approximated" by a sufficiently large finite answer set.

Definition 1.5 (Deterministic Online Algorithm)
*A **deterministic online algorithm** ALG for the request-answer game $(R, \mathcal{A}, \mathcal{C})$ is a sequence of functions $f_j \colon R^j \to A_j$, $j \in \mathbb{N}$. The **output** of ALG on the input request sequence $\sigma = r_1, \ldots, r_n$ is*

$$\mathrm{ALG}[\sigma] := (a_1, \ldots, a_m) \in A_1 \times \cdots \times A_m, \quad \text{where } a_j := f_j(r_1, \ldots, r_j).$$

*The **cost** incurred by ALG on σ, denoted by $\mathrm{ALG}(\sigma)$ is defined as*

$$\mathrm{ALG}(\sigma) := C_m(\sigma, \mathrm{ALG}[\sigma]).$$

We now define the notion of a randomized online algorithm.

Definition 1.6 (Randomized Online Algorithm)
*A **randomized online algorithm** RALG is a probability distribution over deterministic online algorithms ALG_x (x may be thought of as the coin tosses of the algorithm RALG). The answer sequence $\mathrm{RALG}[\sigma]$ and the cost $\mathrm{RALG}(\sigma)$ on a given input σ are random variables.*

In terms of game theory we have defined a randomized algorithm as a *mixed strategy* (where a deterministic algorithm is then considered to be a *pure strategy*). There is no harm in using this definition of a randomized algorithm since without memory restrictions all types

[1] It is also possible to define online profit-maximization problems. For those problems, the serving of each request yields a profit and the goal is to maximize the total profit obtained.

of randomized strategies, *mixed strategies*, *behavioral strategies*, and *general strategies*, are equivalent (see e.g. [6, Chapter 6]).

We illustrate request-answer games by two simple examples. The first example is the classical *paging problem*, which we will study in more detail in Chapter 3.

Example 1.7 (Paging Problem)
Consider a two-level memory system (e.g., of a computer) that consists of a small fast memory (the cache) with k pages and a large slow memory consisting of a total of N pages. Each request specifies a page in the slow memory, that is, $r_j \in R := \{1,\ldots,N\}$. In order to serve the request, the corresponding page must be brought into the cache. If a requested page is already in the cache, then the cost of serving the request is zero. Otherwise one page must be evicted from the cache and replaced by the requested page at a cost of 1. A paging algorithm specifies which page to evict. Its answer to request r_j is a number $a_j \in A_j := \{1,\ldots,k\}$, where $a_j = p$ means to evict the page at position p in the cache.

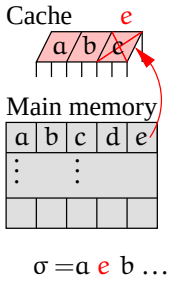

Figure 1.4: Paging problem with cache size $k = 3$.

The objective in the paging problem is to minimize the number of page faults. The cost function C_j simply counts the number of page faults which can be deduced easily from the request sequence $r_1,\ldots,r_j$ and the answers $a_1,\ldots,a_j$. ◁

In the paging problem the answer to a request implies an irrevocable decision of how to serve the next request, that is, the paging problem is formulated within the sequence model. We will now provide a second example of a request-answer game which specifies a problem in the time stamp model, where decisions can be revoked as long as they have not been executed yet.

Example 1.8 (Online TCP Acknowledgment)
In the online TCP acknowledgment problem introduced in [8] a number of *packets* are received. Each packet j has a receipt time $t_j \geqslant 0$ and a weight $w_j \geqslant 0$. An online algorithm learns the existence of a packet only at its receipt time.

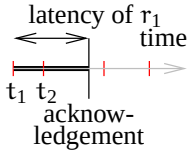

Figure 1.5: Online TCP acknowledgement problem.

All packets must be acknowledged at some time after their receipt. A single acknowledgment acknowledges all packets received since the last acknowledgment. There is a cost of 1 associated with each acknowledgment. Moreover, if packet j is acknowledged

at time $t \geqslant t_j$, this induces a latency cost of $w_j(t - t_j)$. The goal in the online TCP acknowledgment problem is to minimize the sum of the acknowledgment cost and the latency cost.

At time t_j when packet $r_j = (t_j, w_j)$ is received, an online algorithm ALG must decide when to acknowledge all yet unconfirmed packets. Hence, the answer to request $r_j \in R := \mathbb{R}_+ \times \mathbb{R}_+$ is a real number $a_j \in A_j := \mathbb{R}_+$ with $a_j \geqslant t_j$. If an additional packet is received before time a_j, then ALG is not charged for the intended acknowledgment at time a_j, otherwise it incurs an acknowledgment cost of one. The cost function C_j counts the number of actual acknowledgments and adds for each packet r_j the latency cost resulting from the earliest realized acknowledgment in the answer sequence. The condition that packets can not be acknowledged before they are received can be enforced by defining the value $C_j(r_1, \ldots, r_j, a_1, \ldots, a_j)$ to be $+\infty$ if $a_j < t_j$. ◁

1.3 Competitive Analysis

We now define competitiveness of deterministic and randomized algorithms. While the deterministic case is straightforward, the randomized case is much more subtle.

1.3.1 Deterministic Algorithms

Suppose that we are given an online problem as a request-answer game. We define the *optimal offline cost* on a sequence $\sigma \in R^m$ as follows:

$$\mathrm{OPT}(\sigma) := \min\{ C_m(\sigma, a) : a \in A_1 \times \cdots \times A_m \}.$$

The optimal offline cost is the yardstick against which the performance of a deterministic algorithm is measured in competitive analysis.

Definition 1.9 (Deterministic Competitive Algorithm)
Let $c \geqslant 1$ be a real number. A deterministic online algorithm ALG *is called* c-***competitive*** *if, there exists a constant b such that*

$$\mathrm{ALG}(\sigma) \leqslant c \cdot \mathrm{OPT}(\sigma) + b \tag{1.1}$$

holds for any request sequence σ. The ***competitive ratio*** *of* ALG *is the infimum over all c such that* ALG *is c-competitive.*

Observe that, in the above definition, there is no restriction on the computational resources of an online algorithm. The only scarce resource in competitive analysis is information.

We want to remark here that the definition of c-competitiveness varies in the literature. Some authors allow b to depend on some problem or instance specific parameters and many other authors leave the constant b out of the definition, i.e., $b = 0$. This last case is also known as *strict competitiveness*.

Since for a c-competitive algorithm ALG we require inequality (1.1) to hold for *any* request sequence, we may assume that the request sequence is generated by a malicious *adversary*. Competitive analysis can be thought of as a game between an *online player* (an online algorithm) and the adversary. The adversary knows the (deterministic) strategy of the online player, and can construct a request sequence which maximizes the ratio between the player's cost and the optimal offline cost.

Example 1.10 (A Competitive Algorithm for the Ski Rental Problem)
Due to the simplicity of the Ski Rental Problem *all possible* deterministic online algorithms can be specified. A generic online algorithm ALG_j rents skis until the woman has skied $j-1$ times for some $j \geqslant 1$ and then buys skis on day j. The value $j = \infty$ is allowed and means that the algorithm never buys. Clearly, each such algorithm is online. Notice that on a specific request sequence σ algorithm ALG_j might not get to the point that it actually buys skis, since σ might specify less than j skiing days. We claim that ALG_j for $j = B$ is c-competitive with $c = 2-1/B$.

Let σ be any request sequence specifying n skiing days. Then our algorithm has cost $\text{ALG}_B(\sigma) = n$ if $n \leqslant B-1$ and cost $\text{ALG}_B(\sigma) = B-1+B = 2B-1$ if $n \geqslant B$. Since the optimum offline cost is given by $\text{OPT}(\sigma) = \min\{n, B\}$, it follows that our algorithm is $(2-1/B)$-competitive. ◁

Example 1.11 (Lower Bounds for the Ski Rental Problem)
In Example 1.10 we derived a $(2-1/B)$-competitive algorithm for the ski rental problem with buying cost B. Can we do any better?

The answer is »no«! Any competitive algorithm ALG must buy skis at some point in time, say, day j. The adversary simply presents skiing requests until the algorithm buys and then ends the sequence. Thus, the online cost is $\text{ALG}(\sigma) = j-1+B$, whereas the optimal offline cost is $\text{OPT}(\sigma) = \min\{j, B\}$.

If $j \leqslant B$, then

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} = \frac{j-1+B}{\min\{j,B\}} = \frac{j-1+B}{j} = 1+\frac{B-1}{j} \geqslant 1+\frac{B-1}{B} = 2-\frac{1}{B}.$$

If $j \geqslant B$, then

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} = \frac{j-1+B}{\min\{j,B\}} \geqslant \frac{2B-1}{B} = 2-\frac{1}{B}.$$

Hence, the ratio of the online cost and the offline cost is at least $2-1/B$. ◁

1.3.2 Randomized Algorithms

For randomized algorithms we have to be precise in defining what kind of information about the online player is available to the adversary. This leads to different adversary models which are explained below. For an in-depth treatment we refer to [6, 23].

If the online algorithm is randomized then according to intuition, the adversary has less power since the moves of the online player are no longer certain. The weakest kind of adversary for randomized algorithms is the *oblivious adversary*:

Definition 1.12 (Oblivious Adversary)
*An **oblivious adversary** OBL must construct the request sequence in advance based only on the description of the online algorithm but before any moves are made.*

We can now define competitiveness against this adversary:

Definition 1.13 (Competitive Algorithm against an Oblivious Adversary)
*A randomized algorithm RALG, distributed over a set $\{\text{ALG}_y\}$ of deterministic algorithms, is c-**competitive against an oblivious adversary** for some $c \geqslant 1$, if*

$$\mathbb{E}_Y[\text{ALG}_y(\sigma)] \leqslant c \cdot \text{OPT}(\sigma).$$

for all request sequences σ. Here, the expression $\mathbb{E}_Y[\text{ALG}_y(\sigma)]$ denotes the expectation with respect to the probability distribution Y over $\{\text{ALG}_y\}$ which defines RALG.

In case of a deterministic algorithm the above definition collapses to that given in Definition 1.9.

Example 1.14 (Ski Rental Problem Revisited)
We look again at the Ski Rental Problem given in Example 1.1. We now consider the following randomized algorithm RANDSKI against an oblivious adversary. Let $\rho := B/(B-1)$ and $\alpha := \frac{\rho-1}{\rho^B-1}$. At the start RANDSKI chooses a random number $k \in \{0,\ldots,B-1\}$ according to the distribution $\Pr[k=x] := \alpha\rho^k$. After that, RANDSKI works completely deterministic, buying skis after having skied k times. We analyze the competitive ratio of RANDSKI against an oblivious adversary. Note that it suffices to consider sequences σ specifying at most B days of skiing. For a sequence σ with $n \leqslant B$ days of skiing, the optimal cost is clearly $\mathrm{OPT}(\sigma) = n$. The expected cost of RANDSKI can be computed as follows

$$\mathbb{E}[\textsc{randski}(\sigma)] = \sum_{k=0}^{n-1} \alpha\rho^k(k+B) + \sum_{k=n}^{B-1} \alpha\rho^k n.$$

We now show that $\mathbb{E}[\textsc{randski}(\sigma)] = c_\rho := \frac{\rho^B}{\rho^B-1}\mathrm{OPT}(\sigma)$.

$$\begin{aligned}
\mathbb{E}[\textsc{randski}(\sigma)] &= \sum_{j=0}^{n-1} p_j(j+B) + \sum_{j=n}^{B-1} p_j n \\
&= \sum_{j=0}^{n-1} \alpha\rho^j(j+B) + \sum_{i=n}^{B-1} \alpha\rho^i n \\
&= \alpha\sum_{i=0}^{n-1} \rho^j j + \alpha B\sum_{j=0}^{n-1} \rho^j + \alpha n\sum_{j=n}^{B-1} \rho^j \\
&\leqslant \alpha\frac{(n-1)\rho^{n+1} - n\rho^n + \rho}{(\rho-1)^2} + \alpha B\frac{\rho^n-1}{\rho-1} + \alpha n\frac{\rho^B-\rho^n}{\rho-1}
\end{aligned}$$

(use $1/B = \rho-1$)

$$\begin{aligned}
&= \frac{\alpha}{(\rho-1)^2}\left((n-1)\rho^{n+1} - n\rho^n + \rho + \rho^n - 1 + n(\rho-1)(\rho^B-\rho^n)\right) \\
&= \frac{\alpha}{(\rho-1)^2}\left(n\rho^{B+1} - n\rho^B - \rho^{n+1} + \rho^n + \rho - 1\right) \\
&= \frac{\alpha}{(\rho-1)^2}\left((\rho-1)n\rho^B - (\rho-1)(-\rho^n+1)\right) \\
&= \frac{\alpha}{\rho-1}\left(n\rho^B - \rho^n + 1\right) \\
&\leqslant \frac{\alpha}{\rho-1}\cdot n\rho^B \qquad \text{(since } \rho \geqslant 1\text{)} \\
&= \frac{\rho^B}{\rho^B-1}\cdot n \\
&= \frac{\rho^B}{\rho^B-1}\cdot \mathrm{OPT}(\sigma).
\end{aligned}$$

Hence, RANDSKI is c_B-competitive with $c_B = \frac{\rho^B}{\rho^B-1}$. Since $\lim_{B\to\infty} c_B = e/(e-1) \approx 1.58$, this algorithm achieves a better competitive ratio than any deterministic algorithm whenever $2B-1 > e/(e-1)$, that is, when $B > (2e-1)/2(e-1)$. ◁

In contrast to the oblivious adversary, an *adaptive adversary* can issue requests based on the online algorithm's answers to previous ones.

The *adaptive offline adversary* (ADOFF) defers serving the request sequence until he has generated the last request. He then uses an optimal offline algorithm. The *adaptive online adversary* (ADON) must serve the input sequence (generated by himself) online. Notice that in case of an adaptive adversary ADV, the adversary's cost $\text{ADV}(\sigma)$ for serving σ is a random variable.

Definition 1.15 (Competitive Algorithm against an Adaptive Adversary)
A randomized algorithm RALG*, distributed over a set* $\{\text{ALG}_y\}$ *of deterministic algorithms, is called* c-***competitive against an adaptive adversary*** ADV *for some* $c \geqslant 1$*, where* $\text{ADV} \in \{\text{ADON}, \text{ADOFF}\}$*, if*

$$\mathbb{E}_Y[\text{ALG}_y(\sigma) - c \cdot \text{ADV}(\sigma)] \leqslant 0.$$

for all request sequences σ*. Here,* $\text{ADV}(\sigma)$ *denotes the adversary cost which is a random variable.*

The above adversaries differ in their power. Clearly, an oblivious adversary is the weakest of the three types and an adaptive online adversary is no stronger than the adaptive offline adversary. Moreover, as might be conjectured, the adaptive offline adversary is so strong that randomization adds no power against it. More specifically, the following result holds:

Theorem 1.16 ([4]) *Let* $(R, \mathcal{A}, \mathcal{C})$ *be a request-answer game. If there exists a randomized algorithm for* $(R, \mathcal{A}, \mathcal{C})$ *which is* c*-competitive against any adaptive offline adversary, then there exists also a* c*-competitive deterministic algorithm for* $(R, \mathcal{A}, \mathcal{C})$*.* □

The strength of the adaptive online adversary can also be estimated:

Theorem 1.17 ([4]) *Let* $(R, \mathcal{A}, \mathcal{C})$ *be a request-answer game. If there exists a randomized algorithm for* $(R, \mathcal{A}, \mathcal{C})$ *which is* c*-competitive against any adaptive online adversary, then there exists a* c^2*-competitive deterministic algorithm for* $(R, \mathcal{A}, \mathcal{C})$*.* □

For more information about the relations between the various adversaries we refer to [4] and the textbooks [6, 23].

1.4 Exercises

Exercise 1.1
Give a definition of a c-competitive algorithm and the competitive ratio for maximization problems. These definitions should be analogue to the definitions for minimization problems: the smaller c is, the better the algorithm.

Exercise 1.2
Given an optimal offline-algorithm for the ice cream problem in Example 1.2.

Exercise 1.3
Formulate the ski rental problem, the ice cream problem as request answer games.

Exercise 1.4
Formulate the elevator problem and the scheduling problem from Example 1.3 as request answer problems (using infinite answer sets).

Exercise 1.5
Show that against an adaptive offline adversary RANDSKI does not achieve a competitive ratio smaller than $2 - 1/B$ without using Theorem 1.16.

Exercise 1.6 (Bin packing)
In *bin packing*, a finite set of items of size $s_i \in (0,1]$ is supposed to be packed into bins of unit capacity using the minimum possible number of bins. In online bin packing, an item i has to be packed before the next item $i+1$ becomes known. Once an item is packed it cannot be removed and put in another bin.

(a) Show that any c-competitive deterministic algorithm for the online Bin Packing Problem has $c \geqslant 4/3$.

(b) The FIRSTFIT-Algorithm puts an item i always in the first bin that has still enough space to fit in i. If there is no bin left with enough space then a new bin is opened. Show that FIRSTFIT is 2-competitive.

Exercise 1.7 (Cow path problem)
A student has parked his car in the Straße des 17. Juni (which is a long street near to the Technical University of Berlin), and he can't remember whether the car is to the left or right of his current place. As it is late at night he can only see the car standing right in front of his nose. How does the student find back his car and walks as little as possible.

We want to help him with that. Thereto, we model the carsearch as the search to an unknown point $a \in \mathbb{R}$. The current place where the student is standing is the origin 0.

The optimal offline strategy know the parking spot of the car, a, and has cost $|a|$ (the length of the shortest path from 0 to a).

(a) Does there exist a strict competitive algorithm?

(b) Design a c-competitive algorithm for a constant c.

Hint: Consider the algorithm that first walks $\alpha > 1$ units to the right, then back to the origin, and then α^2 units to the left. The ith turningpoint is then in point $(-1)^{i+1}\alpha^i$. What is the competitive ratio of this algorithm, and how should we choose α?

2 Examples and Elementary Techniques

In this section we present elementary techniques that are used in the design and analysis of algorithms. Our toy problem, so as to speak, will be the *list accessing problem*. In this problem we are given a linear list L containing a finite number of elements. Requests for elements in the list arrive online. Given a request for an element x the *cost* of answering the request is the position of x in L. In order to minimize the cost, one can reorganize the list as follows:

Free exchanges Directly after a request to x, the element x can be moved to an arbitrary position further to the front of the list at no cost.

Paid exchanges Other exchanges of adjacent elements in the list are possible at unit cost.

The motivation for studying the above cost model is the situation of a linked list data structure. During a search from the front of the list to a specific element x one could store a pointer to an arbitrary position further to the front. Thus, a move of x could be carried out at »no cost«. The goal of the list accessing problem is to minimize the total cost, which equals the sum of the access and reorganization cost. An online algorithm for the list accessing problem must answer request r_i before the next request r_{i+1} is revealed.

We consider the following online strategies:

Algorithm mtf (move to the front) After request r_i the element r_i is moved to the front of the list (using only free exchanges).

Algorithm trans (transpose) After request r_i the element r_i is moved to the front one position by exchanging it with its predecessor in the list (using a free exchange)

Algorithm fc (frequency count) The algorithm maintains for each element in the list L a counter count[x], indicating how often x has been requested. Upon a request r_i the counter count[r_i] is updated and the list is resorted according to decreasing counter values afterwards.

2.1 Cruel Adversaries

The *cruel adversary* concept tries to enforce a bad behavior of an online algorithm by hurting it as much as possible in every single step. We use this adversary for proving that FC is not competitive.

Theorem 2.1 *For any $\varepsilon > 0$ there are arbitrary long request sequences σ such that* $\text{TRANS}(\sigma)/\text{OPT}(\sigma) \geqslant 2m/3 - \varepsilon$.

Proof: The cruel adversary always requests the last element in the list of TRANS. This amounts to requesting the elements that were in the last two positions of the initial list alternatingly. The cost of ALG on such a sequence of length n are $\text{TRANS}(\sigma) = nm$, where as usual m denotes the number of elements in the list.

The adversary can serve the request sequence as follows. After the first two requests, both relevant elements are moved to the first two positions in its list. Hence, from this moment on, the cost of the adversary for any pair of requests to these elements is 3. Thus, the total cost of OPT can be bounded from above as $\text{OPT}(\sigma) \leqslant 2m + 3(n-2)/2$. Comparing this value with the cost of nm for TRANS shows the theorem. □

2.2 Potential Functions

Potential functions are an important tool for proving competitiveness results. In our example, we use such a potential function for the analysis of MTF.

Let ALG be an online algorithm for some online minimization problem. The potential Φ maps the current configurations of ALG and OPT to a nonnegative value $\Phi \geqslant 0$. We denote by Φ_i the potential after request i. Let c_i be the cost incurred by MTF on request r_i. The *amortized cost* a_i for serving request r_i is defined as

$$a_i = \underbrace{c_i}_{\text{cost of MTF for } r_i} + \underbrace{\Phi_i - \Phi_{i-1}}_{\text{change in potential}}.$$

The intuition behind a potential function and the amortized cost is to measure »how well« the current configuration of the online algorithm is compared to an optimal offline algorithm. The potential can be viewed as a bank account. If the difference $\Phi_i - \Phi_{i-1}$ is negative, then the amortized cost underestimate the real cost c_i. The difference is covered by withdrawal from the account. We have:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i + (\Phi_0 - \Phi_n) \leqslant \sum_{i=1}^{n} a_i + \Phi_0. \tag{2.1}$$

Thus, up to an additive constant, the real cost is bounded from above by the amortized cost. If we can show that $a_i \leqslant c\,\text{OPT}(r_i)$, where $\text{OPT}(r_i)$ is the cost incurred by the optimal offline algorithm on request r_i, then it follows that the online algorithm ALG is c-competitive.

In our particular application, we define the potential function by

$$\begin{aligned}\Phi_i &= \text{number of inversions}\\ &= |\{(a,b) : a \text{ is before } b \text{ in MTF's list, but } b \text{ is before } a \text{ in the list of OPT}\}|.\end{aligned}$$

When bounding the amortized cost, we imagine that request r_i is first served by MTF and then by OPT.

Let x be the element requested in r_i, which is at position k in MTF's list and at position j in the list organized by OPT(see Figure 2.1 for an illustration). We denote by f and p the number of free respective paid exchanges used by OPT on this request.

Let v be the number of elements that are in front of x in MTF's list but behind x in OPT's list before the request (these elements are indicated by stars $*$ in Figure 2.1). By moving x to

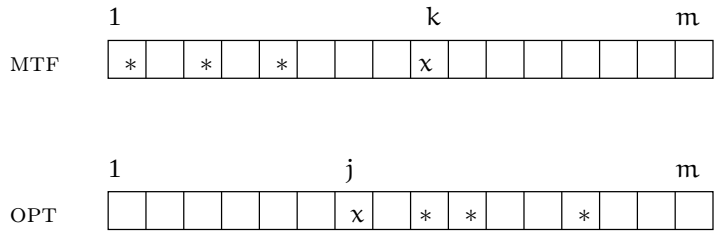

Figure 2.1: Configurations of the lists organized by MTF and OPT at the time of request $r_i = x$.

the front of the list, ν inversions are removed and at most $j-1$ new inversions are created. Thus, the change in potential due to actions of MTF is at most $-\nu+j-1$.

By reorganizing the list, OPT can increase the potential by at most $p-f$ (every free exchange decreases the number of inversions, every paid exchange can increase the number of inversions by at most one). Thus, the amortized cost satisfies:

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= k + \Phi_i - \Phi_{i-1} \\ &\leqslant k - \nu + j - 1 + p - f \\ &= j + p + k - \nu - 1. \end{aligned} \tag{2.2}$$

Recall that ν is the number of elements in MTF's list that were in front of x but came after x in OPT's list. Thus $k-1-\nu$ elements are before x in both lists. Since OPT contained x at position j, we can conclude that $k-\nu-1 \leqslant j-1$. Using this inequality in (2.2) gives

$$a_i \leqslant j + p + (j-1) \leqslant 2(j+p) - 1 = 2\mathrm{OPT}(r_i) - 1.$$

Summing over all requests $r_1, \ldots, r_n$ results in $\sum_{i=1}^{n} a_i \leqslant 2\mathrm{OPT}(\sigma) - n$. Since $\mathrm{OPT}(\sigma) \leqslant nm$, we can conclude that

$$\sum_{i=1}^{n} a_i \leqslant \left(2 - \frac{1}{m}\right) \mathrm{OPT}(\sigma). \tag{2.3}$$

We finally use that MTF and OPT start with identical list configurations such that $\Phi_0 = 0$. From (2.3) and (2.1) we get the following theorem:

Theorem 2.2 MTF *achieves a competitive ratio of* $2-1/m$ *for the list accessing problem.* □

2.3 Averaging over Adversaries

Usually, the optimal offline cost is hard to bound both from above and below. For proving lower bound results on the competitive ratio of online algorithm *averaging over adversaries* can help. The basic idea is to have a set M of algorithms each of which serves the same request sequence σ. If the sum of the costs of all algorithms in M is at most C, then there must be some algorithm in M which has cost at most $C/|M|$, the average cost. Hence, we get that $\mathrm{OPT}(\sigma) \leqslant C/|M|$.

Theorem 2.3 *Any deterministic algorithm for the list accessing problem has a competitive ratio at least* $2-2/(m+1)$*, where* $m = |L|$ *denotes the length of the list.*

Proof: We use a cruel adversary in conjunction with the averaging idea outlined above. Given a deterministic algorithm ALG, the cruel adversary chooses a sequence length n and always requests the last element in ALG's list. The online cost is then $\text{ALG}(\sigma) = nm$.

Consider the $m!$ static offline algorithms which correspond to the $m!$ permutations of the list elements. Each of these algorithm initially sorts the list according to its permutation and then keeps the list fixed for the rest of the sequence. The sorting cost of each algorithm can be bounded by a constant b which depends only on m.

We now bound the sum of the costs of the $m!$ static algorithms on a request $r_i = x$. For each of the m possible positions of x there are exactly $(m-1)!$ permuations which have x at this positions. Hence, the sum of the costs is

$$\sum_{j=1}^{m} j(m-1)! = (m-1)!\frac{m(m+1)}{2}.$$

Thus, the average cost of the static algorithms is at most

$$b + n\frac{m+1}{2}.$$

This yields:

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} \geqslant \frac{nm}{n\frac{m+1}{2}+b} = \frac{2m}{m+1+2b/n} \overset{n\to\infty}{\to} \frac{2m}{m+1} = \left(2-\frac{2}{m+1}\right).$$

This completes the proof. □

2.4 Yao's Principle and Lower Bounds for Randomized Algorithms

A lower bound on the competitive ratio is usually derived by providing a set of specific instances on which no online algorithm can perform well compared to an optimal offline algorithm. Here, again, we have to distinguish between deterministic and randomized algorithms.

For deterministic algorithms finding a suitable set of request sequences is in most cases comparatively easy. For randomized algorithms, however, it is usually very difficult to bound the expected cost of an arbitrary randomized algorithm on a specific instance from below. As an illustration, consider the extremely simple ski rental problem. How would you fool an arbitrary randomized algorithm?

In this section we derive a lower bound technique called *Yao's principle* which has become *the* standard tool for proving lower bounds for randomized algorithms.

Theorem 2.4 (Yao's Principle) *Let $\{\text{ALG}_y : y \in \mathcal{Y}\}$ denote the set of deterministic online algorithms for an online minimization problem. If X is a probability distribution over input sequences $\{\sigma_x : x \in \mathcal{X}\}$ and $\bar{c} \geqslant 1$ is a real number such that*

$$\inf_{y\in\mathcal{Y}} \mathbb{E}_X[\text{ALG}_y(\sigma_x)] \geqslant \bar{c}\,\mathbb{E}_X[\text{OPT}(\sigma_x)], \tag{2.4}$$

then $\bar{c}$ is a lower bound on the competitive ratio of any randomized algorithm against an oblivious adversary.

Yao's principle allows to trade randomization in an online algorithm for randomization in the input. Basically, it says the following: if we have a probability distribution X over the input sequences such that on average (with respect to X) every deterministic algorithm performs badly (compared to an optimal offline algorithm), then also for every randomized algorithm there exists a bad input sequence.

We will prove Yao's principle only in the special case of a finite set of input sequences and a finite set of possible deterministic algorithms. This way we avoid a lot of subtle issues about the existence of expectations and measure theory. Since Yao's principle is derived from game theoretic concepts, we briefly review the necessary material to explain the ideas behind the theorem.

2.4.1 Two-Person Zero-Sum Games

Consider the following children's game: Richard and Chery put their hands behind their backs and make a sign for one of the following: stone (closed fist), paper (open palm), and scissors (two fingers). Then, they simultaneously show their chosen sign. The winner is determined according to the following rules: stone beats scissors, scissors beats paper and paper beats stone. If both players show the same sign, the game is a draw. The loser pays 1 € to the winner. In case of a draw, no money is payed.

We can represent the scissors-paper-stone game as a matrix M, where the rows correspond to Richard's choices and the columns represent the choices of Chery. The entry m_{ij} is the amount of money Richard wins if he chooses i and Chery chooses j. Table 2.1 shows the corresponding matrix.

	Stone	Scissors	Paper
Stone	0	1	-1
Scissors	-1	0	1
Paper	1	-1	0

Table 2.1: Matrix for the scissors-paper-stone game

The scissors-paper-stone game is an example of a two-person zero sum game. The matrix associated with the game is called *payoff matrix* and the game is a *zero-sum game* since the net amount by Richard and Chery is always exactly zero.

Definition 2.5 (Two-person zero-sum game)
Given any matrix $n \times m$-Matrix $M = (m_{ij})$, the $n \times m$ ***two-person zero-sum game*** *Γ_M with payoff matrix M is played by two persons R and C. The set of possible strategies for the* ***row player*** *R is in correspondence with the rows of M, and likewise for the strategies of the column player C. These strategies are called* ***pure*** *strategies of R and C. The entry m_{ij} is the amount paid by C to R when R chooses strategy i and C chooses strategy j. The matrix is known to both players.*

The goal of the row player R is to maximize his profit, while the column player C seeks to minimize her loss.

You may imagine the situation of an online player and the offline adversary in the game theoretic setting as follows[1]: the rows correspond to all possible deterministic online algorithms, the columns are in correspondence with the possible input sequences. If the online player chooses ALG_i and the adversary σ_j, then the payoff of the adversary (who is the row player) is $\mathrm{ALG}_i(\sigma_j)/\mathrm{OPT}(\sigma_j)$.

[1] This imagination is not exactly how we will handle things later, but for the moment it gives a good motivation.

Suppose that R chooses strategy i. Then, no matter what C chooses, he is assured a win of $\min_j m_{ij}$. An *optimal (pure) strategy* for R is to choose i^* such that $\min_j m_{i^*j} = \max_i \min_j m_{ij}$. We set

$$V_R := \max_i \min_j m_{ij} \tag{2.5}$$

$$V_C := \min_j \max_i m_{ij}. \tag{2.6}$$

Here, Equation (2.6) gives the value corresponding to the optimal strategy of the row player. It is easy to see that $V_R \leqslant V_C$ for any two-person zero sum game (see Exercise 2.2). In fact, strict inequality may occur (see Exercise 2.3). If $V_R = V_C$ then the common value is called the *value of the game*.

Now let us do the transition to randomized or *mixed* strategies. A mixed strategy for a player is a probability distribution over his/her deterministic strategies. Thus, a mixed strategy for R is a vector $p = (p_1, \ldots, p_n)^T$ and for C it is a vector $q = (q_1, \ldots, q_m)$ such that all entries are nonnegative and sum to one.

The payoff to R becomes a random variable with expectation

$$\sum_{i=1}^{n} \sum_{j=1}^{m} p_i m_{ij} q_j = p^T M q.$$

How do optimal mixed strategies look like? R can maximize his expected profit by choosing p such as to maximize $\min_q p^T M q$. Likewise, C can choose q to minimize $\max_p p^T M q$. We define

$$v_R := \max_p \min_q p^T M q \tag{2.7}$$

$$v_C = \min_q \max_p p^T M q. \tag{2.8}$$

The minimum and maximum in the above equations are taken over all possible probability distributions.

Observe that once p is fixed in (2.7), then $p^T M q$ becomes a linear function of q and is minimized by setting to one the entry q_j with the smallest coefficient in this linear function. A similar observation applies to (2.8). We thus have

$$v_R = \max_p \min_j p^T M e_j \tag{2.9a}$$

$$v_C = \min_q \max_i e_i^T M q. \tag{2.9b}$$

The famous Minimax Theorem von von Neumann states that with respect to mixed strategies, any two-person zero-sum game has a value:

Theorem 2.6 (von Neumann's Minimax Theorem) *For any two-person zero-sum game specified by a matrix* M

$$\max_p \min_q p^T M q = \min_q \max_p p^T M q. \tag{2.10}$$

A pair of of mixed strategies (p^*, q^*) *which maximizes the left-hand side and minimizes the right-hand side of* (2.10) *is called a* ***saddle-point*** *and the two distributions are called* ***optimal mixed strategies****.*

Proof: By (2.9) it suffices to show that

$$\max_p \min_j p^T M e_j = \min_q \max_i e_i^T M q. \tag{2.11}$$

Consider the left-hand side of (2.11). The goal is to choose p such that $\min_j \sum_{i=1}^{n} m_{ij} p_i$ is maximized. This can be equivalently expressed as the following Linear Program:

$$\max \quad z \tag{2.12a}$$

$$z - \sum_{i=1}^{n} m_{ij} p_i \leqslant 0 \qquad j = 1, \ldots, m \tag{2.12b}$$

$$\sum_{i=1}^{n} p_i = 1 \tag{2.12c}$$

$$p \geqslant 0 \tag{2.12d}$$

Similarly, we can reformulate the right-hand side of (2.11) as another Linear Program:

$$\min \quad w \tag{2.13a}$$

$$w - \sum_{j=1}^{m} m_{ij} q_j \geqslant 0 \qquad i = 1, \ldots, n \tag{2.13b}$$

$$\sum_{j=1}^{m} q_j = 1 \tag{2.13c}$$

$$q \geqslant 0. \tag{2.13d}$$

Observe that (2.12) and (2.13) are dual to each other. Clearly, both Linear Programs are feasible, hence the claim of the theorem follows by the Duality Theorem of Linear Programming. □

Using our observation in (2.9) we can reformulate the Minimax Theorem as follows:

Theorem 2.7 (Loomi's Lemma) *For any two-person zero-sum game specified by a matrix M*

$$\max_p \min_j p^T M e_j = \min_q \max_i e_i^T M q.$$

□

Loomi's Lemma has an interesting consequence. If C knows the distribution p used by R, then her optimal mixed strategy is in fact a pure strategy. The analogous result applies vice versa.

From Loomi's Lemma we can easily derive the following estimate, which is known as Yao's Principle: Let $\bar{p}$ be a fixed distribution over the set of pure strategies of R. Then

$$\begin{aligned} \min_q \max_i e_i^T M q &= \max_p \min_j p^T M e_j \qquad \text{(by Loomi's Lemma)} \\ &\geqslant \bar{p}^T \min_j M e_j. \end{aligned}$$

The inequality

$$\min_q \max_i e_i^T M q \geqslant \bar{p}^T \min_j M e_j \tag{2.14}$$

is referred to as *Yao's Principle*.

2.4.2 Yao's Principle and its Application to Online Problems

We now prove the finite version of Yao's Principle for Online-Problems (cf. Theorem 2.4). Let Π be an online problem with a finite set of deterministic online algorithms $\{\mathrm{ALG}_j : j = 1, \ldots, m\}$ and a finite set $\{\sigma_i : i = 1, \ldots, n\}$ of input instances.

Suppose we want to prove that no randomized algorithm can achieve a competitive ratio smaller than c against an oblivious adversary. Then we need to show that for any distribution q over $\{\text{ALG}_j\}$ there is an input sequence σ_i such that

$$\mathbb{E}_q\left[\text{ALG}_j(\sigma_i)\right] \geqslant c \cdot \text{OPT}(\sigma_i).$$

This is equivalent to proving that

$$\min_q \max_i \left\{\mathbb{E}_q\left[\text{ALG}_j(\sigma_i)\right] - c \cdot \text{OPT}(\sigma_i)\right\} \geqslant 0. \tag{2.15}$$

Consider following two-person zero-sum game with payoff matrix $M = (m_{ij})$, where $m_{ij} := \text{ALG}_j(\sigma_i) - c \cdot \text{OPT}(\sigma_j)$. The online player (column player) wants to minimize her loss, the adversary (row player) wants to maximize his win. We have for any distribution $\bar{p}$ over the set of input instances $\{\sigma_i\}$:

$$\begin{aligned}\min_q \max_i \left\{\mathbb{E}_q\left[\text{ALG}_j(\sigma_i)\right] - c \cdot \text{OPT}(\sigma_i)\right\} &= \min_q \max_i e_i^T M q \\ &\geqslant \bar{p}^T \min_j M e_j \\ &= \min_j \mathbb{E}_{\bar{p}}\left[\text{ALG}_j(\sigma_i)\right] - c \cdot \mathbb{E}_{\bar{p}}\left[\text{OPT}(\sigma_i)\right],\end{aligned} \tag{2.16}$$

where $\mathbb{E}_{\bar{p}}\left[\text{ALG}_j(\sigma_i)\right]$ denotes the expected cost of the deterministic algorithm ALG_j with respect to the distribution $\bar{p}$ over the input. Hence, if $\mathbb{E}_{\bar{p}}\left[\text{ALG}_j(\sigma_i)\right] \geqslant c \cdot \mathbb{E}_{\bar{p}}\left[\text{OPT}(\sigma_i)\right]$ for all j, the term in (2.16) will be nonnegative. This shows (2.15).

Example 2.8
Once more we consider the ski rental problem and show how Yao's Principle can be applied to get a lower bound for randomized algorithms.

Let $B := 10$. We construct a probability distribution over a set of two possible input instances: With probability $1/2$, there will be only one day of skiing, with probability $1/2$, there will be 20 days of skiing.

The expected optimal cost under this distribution is given by

$$\mathbb{E}\left[\text{OPT}(\sigma)\right] = \frac{1}{2} \cdot 1 + \frac{1}{2} 10 = \frac{11}{2}.$$

Let ALG_j be the online algorithm which buys skis on day j. Then we have:

$$\begin{aligned}\mathbb{E}\left[\text{ALG}_j(\sigma)\right] &= \begin{cases} 10 & \text{for } j = 1 \\ \frac{1}{2} \cdot 1 + \frac{1}{2}(j - 1 + 10) & \text{für } j = 2, \ldots, 20. \\ \frac{1}{2} + \frac{1}{2} \cdot 20 & \text{for } j > 20. \end{cases} \\ &\geqslant \frac{1}{2} + \frac{11}{2} = 6.\end{aligned}$$

Hence we obtain that

$$\frac{\mathbb{E}\left[\text{ALG}_j(\sigma)\right]}{\mathbb{E}\left[\text{OPT}(\sigma)\right]} \geqslant \frac{6}{\frac{11}{2}} = \frac{12}{11}$$

for all j. Thus, no randomized algorithm can beat the competitive ratio of $12/11$. ◁

2.5 Exercises

Exercise 2.1 (Potential function and amortised costs)
Recall the application of potential functions and amortized costs in the proof of competitiveness of MTF. Potential functions are a very useful and elegant tool as we want to illustrate with the following examples from the area of data structures.

Consider the data structure D_0 on which n operations can be executed. For $i = 1, \ldots, n$ let c_i denote the cost of an algorithm ALG caused by the ith operation. Let D_i denote the data structure after the ith operation. A *potential function* Φ assigns a real number $\Phi(D_i)$ to a data structure D_i. The *amortized costs* $\hat{c}_i$ for the ith operation are

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

The intuition for a potential function and the amortized costs is the following: $\Phi(D_i)$ measures/estimates, »how well« the current state of the structure is. Imagine $\Phi(D_i)$ as a bank account: if the difference $\Phi(D_i) - \Phi(D_{i-1})$ is negative, then $\hat{c}_i$ underestimates the actual costs c_i. The difference can be balanced by withdrawing the potential-loss/difference from the account.

For the amortized cost holds:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n).$$

In the following we analyze stack operations. A stack is a Last-in-First-Out memory/buffer S, on which the following operations are defined:

- PUSH(S, x) puts the object x on top of the stack.
- POP(S) returns the topmost object of the stack and removes it. (If the stack is empty then the operation is aborted with an error message.)

Both operations cost $\mathcal{O}(1)$ time units. Additionally we allow the operation MULTIPOP(S, k), which removes the topmost k objects from the stack. This operation takes $\mathcal{O}(k)$ units of time.

(a) Apply the common worst case analysis and show how to obtain a time bound of $\mathcal{O}(n^2)$ for a sequence of n stack operations (beginning with an empty stack). Assume that for the MULTIPOP-operations there are always sufficiently many elements on the stack.

(b) Use the potential $\Phi(S) := |S|$ and amortized costs in order to improve the worst case bound and obtain a time bound of $\mathcal{O}(n)$.

Exercise 2.2
Consider an arbitrary two-person zero-sum game defined by the $n \times m$-Matrix M. Let

$$V_R := \max_i \min_j m_{ij}$$
$$V_C := \min_j \max_i m_{ij}.$$

Show that $v_R \leqslant v_C$.

Exercise 2.3
Give a two-person zero-sum game such that $V_R < V_C$.

3 The Paging Problem

The paging problem is one of the fundamental and oldest online problem. In fact, this problem inspired the notion of competitive analysis in [33].

In the paging problem we are given a memory system consisting of two levels. Each level can store memory pages of a fixed size. The second level consists of slow memory which stores a fixed set $P = \{p_1, \ldots, p_N\}$ of N pages. The first level is the fast memory (cache), which can store an arbitrary k element subset of P. Usually $k \ll N$.

The memory system is faced with a sequence $\sigma = r_1, \ldots, r_n$ of page requests. Upon a request to page p_i the page is accessed and must be presented in the cache. If p_i is already in the cache, we have a *cache hit* and the system does not need to do anything. If p_i is not in the cache, then we speak of a *cache miss* and the system must copy p_i into the cache thereby replacing another page in the cache. A paging algorithm must therefore decide which page to evict from the cache upon a page fault. An online algorithm for the paging problem must process request r_i before r_{i+1} is revealed to the algorithm.

In this section we consider a simple cost model for the paging problem, the *page fault model*. Under this model, the goal is simply to minimize the number of page faults. This model can be justified by the fact that a cache hit means fast (almost neglectible) access times, whereas a page fault involves access to the slow memory and evicting operations. Typical applications of the paging problem include the organization of CPU/memory systems and caching of disk accesses.

3.1 An Optimal Offline Algorithm

The paging problem is one of the few online problems where an optimal offline algorithm is known and, moreover, this algorithm has efficient (polynomial) running time. We first consider the offline problem, where we are given a known sequence $\sigma = r_1, \ldots, r_n$ of page requests.

Algorithm lfd (longest forward distance) Upon a page fault the algorithm evicts that page from the cache whose next request is furthest into the future.

Algorithm LFD is clearly an offline algorithm since it requires complete knowledge of future page requests.

Lemma 3.1 *Let* ALG *be a deterministic paging algorithm and let* $\sigma = r_1,\ldots,r_n$ *be an arbitrary request sequence. Then for* $i = 1,\ldots,n$ *there is an algorithm* ALG_i *with the following properties:*

(i) ALG_i *serves the first* $i-1$ *requests in the same way as* ALG.

(ii) If the i*th request amounts to a page fault, then* ALG_i *evicts that page from the cache whose next request is furthest into the future (that is, using the* LFD*-rule).*

(iii) $\text{ALG}_i(\sigma) \leqslant \text{ALG}(\sigma)$.

Proof: We construct ALG_i from ALG. If the ith request does not involve a page fault, then there is nothing to prove. Thus, assume for the remainder of the proof that $r_i = f$ produces a page fault for ALG.

We let p be the page which is replaced by ALG by f and q the page which would have been replaced according to the LFD-rule. It suffices to treat the case $p \neq q$.

Algorithm ALG_i replaces q by f. The contents of the cache after r_i are $X \cup \{p\}$ for ALG_i and $X \cup \{q\}$ for ALG, where X is a set of $k-1$ common pages in the cache. Note that $f \in X$ and that the next request to page p must occur before the next request to q, since the LFD-rule caused q to be ejected and not p.

After request r_i the algorithm ALG_i processes the remaining requests as follows until page p is requested again (page p must be requested again, since q is requested after p and we assumed that $p \neq q$).

- If ALG evicts q from the cache, then ALG_i evicts p.
- If ALG evicts $x \neq q$ from the cache, then ALG_i also evicts x.

It is easy to see that until ALG evicts q the caches of the two algorithms are of the form $X' \cup \{p\}$ and $X' \cup \{q\}$ respectively. Thus, the first case causes both caches to unify. From this moment on, both algorithms work the same and incur the same number of page faults. Hence, if the first case occurs before the next request to p there is nothing left to show.

Upon the next request to p (which as we recall once more must occur before the next request to q), ALG incurs a page fault, while ALG_i has a cache hit. If ALG ejects q from the cache, then the caches unify and we are done. Otherwise, the next request to q causes ALG_i to have a page fault whereas ALG has a cache-hit. At this moment ALG_i ejects q and both caches unify again. Clearly, the number of page faults of ALG_i is at most that of ALG. □

Corollary 3.2 LFD *is an optimal offline algorithm for paging.*

Proof: Let OPT be an optimal offline algorithm and consider any request sequence $\sigma = r_1,\ldots,r_n$. Lemma 3.1 gives us an algorithm OPT_1 with $\text{OPT}_1(\sigma) \leqslant \text{OPT}(\sigma)$. We re-apply the Lemma with $i = 2$ to OPT_1 and obtain OPT_2 with $\text{OPT}_2(\sigma) \leqslant \text{OPT}_1(\sigma) \leqslant \text{OPT}(\sigma)$. Repeating this process yields OPT_n, which serves σ just as LFD. □

Lemma 3.3 *Let* $N = k+1$. *Then we have* $\text{LFD}(\sigma) \leqslant \lceil |\sigma|/k \rceil$, *that is,* LFD *has at most one page fault for every* k*th request.*

Proof: Suppose that LFD has a page fault on request r_i and ejects page p of its cache X. Since $N = k+1$, the next page fault of LFD must be on a request to p. When this request occurs, due to the LFD-rule all $k-1$ pages of $X \setminus \{p\}$ must have been requested in between. □

3.2 Deterministic Algorithms

An online algorithm must decide which page to evict from the cache upon a page fault. We consider the following popular caching strategies:

fifo (First-In/First-Out) Eject the page which has been in the cache for the longest amount of time.

lifo (Last-In/First-Out) Eject the page which was brought into cache most recently.

lfu (Least-Frequently-Used) Eject a page which has been requested least frequently.

lru (Least-Recently-Used) Eject a page whose last request is longest in the past.

We first address the competitiveness of LIFO.

Lemma 3.4 LIFO *is not competitive.*

Proof: Let $p_1,\ldots,p_k$ be the initial cache content and p_{k+1} be an additional page. Fix $\ell \in \mathbb{N}$ arbitrary and consider the following request sequence:

$$\begin{aligned}\sigma &= p_1,p_2,\ldots,p_k,(p_{k+1},p_k)^\ell \\ &= p_1,p_2,\ldots,p_k,\underbrace{(p_{k+1},p_k),\ldots,(p_{k+1},p_k)}_{\ell \text{ times}}\end{aligned}$$

Starting at the $(k+1)$st request, LIFO has a page fault on every request which means $\text{LIFO}(\sigma) = 2\ell$. Clearly, $\text{OPT}(\sigma) = 1$. Since we can choose ℓ arbitrarily large, there are no constants c and α such that $\text{LIFO}(\sigma) \leqslant c\ \text{OPT}(\sigma) + \alpha$ for every sequence. □

The strategy LFU is not competitive either:

Lemma 3.5 LFU *is not competitive.*

Proof: Let $p_1,\ldots,p_k$ be the initial cache content and p_{k+1} be an additional page. Fix $\ell \in \mathbb{N}$ arbitrary and consider the request sequence:

$$\sigma = p_1^\ell,\ldots,p_{k-1}^\ell,(p_k,p_{k+1})^\ell.$$

After the first $(k-1)\ell$ requests LFU incurs a page fault on every remaining request. Hence, $\text{LFU}(\sigma) \geqslant 2(\ell-1)$. On the other hand an optimal offline algorithm can evict p_1 upon the first request to p_{k+1} and thus serve the sequence with one page fault only. □

3.2.1 Phase Partitions and Marking Algorithms

To analyze the performance of LRU and FIFO we use an important tool which is known as k-*phase partition*. Given a request sequence σ, phase 0 of is defined to be the empty sequence. Phase $i+1$ consists of the maximal subsequence of σ after phase i such that the phase contains requests to at most k different pages.

$$\sigma = \overbrace{r_1,\ldots,r_{j_1}}^{k \text{ different pages}},\ \overbrace{r_{j_1+1},\ldots,r_{j_2}}^{k \text{ different pages}},\ldots,\ \overbrace{r_{j_r+1},\ldots,r_n}^{\leqslant k \text{ different pages}}$$

The k-phase partition is uniquely defined and independent of any algorithm. Every phase, except for possibly the last phase, contains requests to exactly k different pages.

Given the k-phase partition we define a marking of all pages in the universe as follows. At the beginning of a phase, all pages are unmarked. During a phase, a page is marked upon the first request to it. Notice that the marking does not depend on any particular algorithm. We call an algorithm a *marking algorithm* if it never ejects a marked page.

Theorem 3.6 *Any marking algorithm is* k*-competitive for the paging problem with cache size* k*.*

Proof: Let ALG be a marking algorithm. We consider the k-phase partition of the input sequence σ. ALG incurs at most k page faults in a phase: upon a fault to page f, the marked page is brought into the cache. Since ALG never evicts a marked page, it can incur at most k page faults in a single phase.

We slighty modify our partition of the input sequence to show that an optimal offline algorithm must incur as many page faults as there are phases in the partition. Segment i of the input sequence starts with the second request in phase i and ends with the first request of phase $i+1$. The last request in segment i is a request that has not been requested in phase i, since otherwise phase i would not be maximal.

At the end of segment i, the cache of OPT contains the first request of phase $i+1$, say p. Until the end of phase i there will be requests to $k-1$ other pages, and, until the end of the segment requests to k other pages. This means that there will be requests to $k+1$ pages until the end of segment i. Since OPT can only keep k pages in the cache, it must incur a page fault. □

Lemma 3.7 LRU *is a marking algorithm.*

Proof: Suppose that LRU ejects a marked page p during a phase on request r_i. Consider the first request r_j $(j < i)$ to p during the phase which caused p to be marked. At this point in time p is the page in the cache of LRU whose last request has been most recently. Let $Y = X \cup \{p\}$ denote the cache contents of LRU after request r_j. In order to have LRU eject p, all other pages in X must have been requested during the phase. Thus, there have been requests to all k pages in Y during the phase. Clearly, $r_i \notin Y$, since otherwise LRU would not incur a page fault on request r_i. But this means that during the phase $k+1$ different pages have been requested. □

Corollary 3.8 LRU *is* k*-competitive for the paging problem with cache size* k*.*

In contrast, FIFO is not a marking algorithm. Nevertheless, FIFO can be shown to be competitive by a slight modification of the proof of Theorem 3.6

Theorem 3.9 FIFO *is* k*-competitive for the paging problem with cache size* k*.*

Proof: See Exercise 3.1. □

3.2.2 A Lower Bound for Deterministic Algorithms

Theorem 3.10 *Let* ALG *be any deterministic online algorithm for the paging problem with cache size* k*. If* ALG *is* c*-competitive, then* $c \geqslant k$*.*

Proof: Let $S = \{p_1, \ldots, p_k, p_{k+1}\}$ be an arbitary $(k+1)$ element subset of pages. We use only requests to pages in S. Without loss of generality we assume that the initial cache contents is $\{p_1, \ldots, p_k\}$.

The request sequence is defined inductively such that ALG has a page fault on every single request. We start with $r_1 = p_{k+1}$. If upon request r_i ALG ejects page p, then $r_{i+1} := p$. Clearly, $\mathrm{ALG}(\sigma) = |\sigma|$. On the other hand, by Lemma 3.3 $\mathrm{OPT}(\sigma) \leqslant |\sigma|/k$ which proves the claim. □

3.3 Randomized Algorithms

In this section we present a randomized algorithm which beats the lower bound for deterministic algoirthms stated in Theorem 3.10. More specificially, it achieves a competitive ratio of $2H_k$, where

$$H_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$$

denotes the k*th Harmonic number*. Observe that $\ln k < H_k \leqslant 1 + \ln k$. Hence, randomization gives an exponential boost in competitiveness for the paging problem.

randmark Initially, all pages are unmarked.

Upon a request to a page p which is not in the cache, p is brought marked into the cache. The page to be evicted is chosen uniform at random anomg all unmarked pages in the cache (if all pages in the cache are already marked, then all marks are erased first).

Upon a request to a page p which is already in the cache, p is marked.

Theorem 3.11 RANDMARK *is* $2H_k$*-competitive against an oblivious adversary.*

Proof: We can assume that RANDMARK and the adversary start with the same cache contents. For the analysis we use again the k-phase partition. Observe that RANDMARK never evicts a marked page and that a marked page remains in the cache until the end of the phase. Hence, RANDMARK deserves its name (it is a randomized marking algorithm). If P_i denotes the pages requested in phase i, then the cache contents of RANDMARK at the end of the phase is exactly P_i.

We call a page that is requested during a phase and which is not in the cache of RANDMARK at the beginning of the phase a *new page*. All other pages requested during a phase will be called *old pages*.

Consider an arbitrary phase i and denote by m_i the number of new pages in the phase. RANDMARK incurs a page fault on every new page. Notice that RANDMARK has *exactly* m_i faults on new pages since a marked page is never evicted until the end of the phase. We now compute the expected number of faults on old pages.

There will requests to at most $k - m_i$ different old pages in a phase (the last phase might not be complete). Let ℓ denote the number of new pages requested before the jth request to an old page. When the jth old page is requested, RANDMARK has evicted $k-(j-1)$ ummarked old pages from the cache and these have been chosen uniformly at random. Hence, the probability that the jth page is not in the cache is $\ell/(k-(j-1)) \leqslant m_i/(k-j+1)$. Hence, the number X_i of page faults in phase i satisfies:

$$\mathbb{E}[X_i] \leqslant m_i + \sum_{j=1}^{k-m_i} \frac{m_i}{k-j+1} = m_i(H_k - H_{m_i} + 1) \leqslant m_i H_k.$$

By linearity of expectation $\mathbb{E}[\text{RANDMARK}\sigma] = \sum_i \mathbb{E}[X_i]$.

We complete the proof by showing that $\text{OPT}(\sigma) \geqslant \sum_i m_i/2$. Together phase $i-1$ and phase i contain requests to at least $k + m_i$ different pages. Since OPT can keep only k pages in the cache, it will incur at least $k + m_i$ page faults during these two phases. We thus get $\text{OPT}(\sigma) \geqslant \sum_i m_{2i}$ and $\text{OPT}(\sigma) \geqslant \sum_i m_{2i+1}$, which amounts to

$$\text{OPT}(\sigma) \geqslant \frac{1}{2}\left(\sum_i m_{2i} + \sum_i m_{2i+1}\right) = \sum_i \frac{m_i}{2}.$$

This proves the claim. □

We now address the question how well randomized algorithms can perform.

Theorem 3.12 *Any randomized algorithm for the paging problem with cache size* k *has competitive ratio at least* H_k *against an oblivious adversary.*

Proof: Our construction uses a subset of the pages P of size $k+1$ consisting of the initial cache contents and one additional page. Let $\bar{p}$ be a distribution over the input sequences with the property that the ℓth request is drawn uniformly at random from P independent of all previous requests.

Clearly, any deterministic paging algorithm faults on each request with probability $1/(k+1)$. Hence, we have

$$\mathbb{E}_{\bar{p}}[\mathrm{ALG}(\sigma^n)] = \frac{n}{k+1}. \tag{3.1}$$

where σ^n denotes request sequences of length n. By Yao's Principle the number

$$r := \lim_{n\to\infty} \frac{n}{(k+1)\mathbb{E}_{\bar{p}}[\mathrm{OPT}(\sigma^n)]}$$

is a lower bound on the competitive ratio of any randomized algorithm against an oblivious adversary. Thus, our goal is to show that $r \geqslant H_k$. The desired inequality will follow if we can show that

$$\lim_{n\to\infty} \frac{n}{\mathbb{E}_{\bar{p}}[\mathrm{OPT}(\sigma^n)]} = (k+1)H_k. \tag{3.2}$$

We have a closer look at the optimal offline algorithm. We partition the sequence σ^n into stochastic phases as follows: the partition is like the k-phase partition with the sole difference that the starts/ends of the phases are given by random variables.

Let X_i, $i = 1,2,\ldots$ be the sequence of random variables where X_i denotes the number of requests in phase i. Notice that the X_i are all independent and uniformly distributed. The jth phase starts with request $r_{S_j-1}+1$, where $S_j := \sum_{i=1}^{j-1} X_j$. Let us denote by $N(n)$ the number of complete phases:

$$N(n) := \max\{j : S_j \leqslant n\}.$$

Our proof of (3.2) proceeds in three steps:

(i) We show that

$$\mathrm{OPT}(\sigma^n) \leqslant N(n)+1. \tag{3.3}$$

(ii) We prove that

$$\lim_{n\to\infty} \frac{n}{\mathbb{E}_{\bar{p}}[N(n)]} = \mathbb{E}_{\bar{p}}[X_i]. \tag{3.4}$$

(iii) We establish that the expected value $\mathbb{E}[X_i]$ satisfies:

$$\mathbb{E}_{\bar{p}}[X_i] = (k+1)H_k - 1. \tag{3.5}$$

The inequalities (3.3)– (3.5) then imply (3.2).

(i) At the end of phase j the cache of the optimal offline algorithm OPT consists of all pages consisted in the phase (with the possible exception of the last phase which might be incomplete). Phase $j+1$ starts with a page fault. If OPT now removes the page from the cache which is requested in the first request of phase $j+2$, then each phase will be served with at most one page fault (this argument heavily uses the fact that we are working on a subset of $k+1$ pages).

(ii) The family of random variables $\{N(n) : n \in \mathbb{N}\}$ forms a renewal process. Equation (3.4) is a consequence of the *Renewal Theorem* (see [6, Appendix E]).

(iii) Our problem is an application of the *coupon collector's problem* (see Exercise 3.2). We are given $k+1$ coupons (corresponding to the pages in P) and a phase ends one step, before we have collected all $k+1$ coupons. Thus, the expected value $\mathbb{E}[X_i]$ is one less than the expected number of rounds in the coupon collectors problem, which by Expercise 3.2 is $(k+1)H_k$

□

3.4 The k-Server Problem

The famous k-*server problem* is a natural generalization of the paging problem. Let (X, d) be a *metric space*, that is, a set X endowed with a metric d satisfying:

$$\begin{aligned} d(x,y) = 0 &\iff x = y \\ d(x,y) &= d(y,x) \\ d(x,z) &\leqslant d(x,y) + d(y,z). \end{aligned}$$

In the k-server problem an algorithm moves k mobile servers in the metric space. The algorithm gets a sequence $\sigma = r_1, \ldots, r_n$ of requests, where each request r_i is a point in the metric space X. We say that a request r_i is answered, if a server is placed on r_i. The algorithm must answer all requests in σ by moving the servers in the metric space. His cost $\text{ALG}(\sigma)$ is the total distance travelled by its servers.

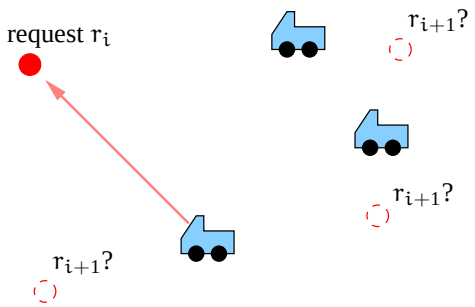

Figure 3.1: When request r_i is issued, one server must be moved to r_i. Only then the next request r_{i+1} becomes known to an online algorithm.

The paging problem is the special case of the k-server problem when $d(x,y) = 1$ for all $x, y \in X$. The k servers represent the k pages in cache, $N = |X|$ is the total number of pages in the system.

This observation already shows that for *some* metric spaces there can be no deterministic algorithm for the k-server problem with competitive ratio $c < k$. We will show now that this result holds true for *any* metric space with at least $k+1$ points.

We call an algorithm *lazy*, if at any request it moves at most one server, and this only if the point requested is not already occupied by one of its servers.

Lemma 3.13 *Let* ALG *be any algorithm for the* k*-server problem. There is a lazy algorithm* ALG*' such that* $\text{ALG}'(\sigma) \leqslant \text{ALG}(\sigma)$ *for any request sequence* σ*. If* ALG *is an online algorithm, then* ALG*' is also an online algorithm.*

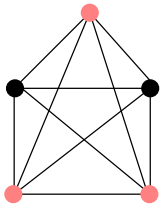

Figure 3.2: Modelling the paging problem with $N = 5$ and $k = 2$ as a k-server problem

Proof: The proof is obtained by a simple induction in conjunction with the triangle inequality.

The modified algorithm ALG' works as follows: suppose that ALG is lazy until request i and then unnecessarily moves a server s from x to y. The cost incurred is $d(x,y)$. The new algorithm ALG' works just as ALG until request i but then skips the movement of s. Let z be the point of the next request served by the server s of ALG. Then, ALG' directly moves the server s from x to z. Since $d(x,z) \leqslant d(x,y) + d(y,z)$ the cost of ALG' is not larger than the one incurred by ALG. □

Lemma 3.13 allows us to restrict ourselves to lazy online algorithm when proving lower bounds.

Theorem 3.14 *Let $M = (X,d)$ be any metric space with $|X| \geqslant k+1$. Then, any deterministic c-competitive algorithm for the k-server problem in M satisfies $c \geqslant k$.*

Proof: Let ALG be any deterministic c-competitive algorithm. By Lemma 3.13 we can assume that ALG is lazy. We show that there are arbitrary long request sequences such that $\text{ALG}(\sigma) \geqslant k\,\text{OPT}(\sigma)$.

We construct σ and k algorithms $B_1,\ldots,B_k$, such that $\text{ALG}(\sigma) = \sum_{j=1}^{k} B_j(\sigma)$. By averaging there must be a B_{j_0} with $B_{j_0}(\sigma) \leqslant k\text{ALG}(\sigma)$.

Let S be the set of points occupied initially by ALG's servers plus an additional point in X. We can assume that $|S| = k+1$. Our sequences only use requests to points in S.

The construction of the sequence $\sigma = r_1, r_2, \ldots$ is inductively. The first request r_1 will be to that point in S where ALG does not have a server. Suppose that at request r_i ALG moves a server from x to r_i. Then, the next request r_{i+1} will be to x. The cost of ALG on σ is thus

$$\text{ALG}(\sigma) = \sum_{i=1}^{n} d(r_{i+1}, r_i) = \sum_{i=1}^{n} d(r_i, r_{i+1}). \tag{3.6}$$

Let $x_1,\ldots,x_k$ be the points in S initially occupied by the servers operated by ALG. For $j = 1,\ldots,k$ algorithm B_j starts with servers in all points except for x_j. If at some point in time a point r_i is requested where B_j does not have a server, it moves a server from r_{i-1} to r_i.

Denote by S_j the points where B_j has its servers. Inititially, all S_j are different. Suppose that these sets are still different until request r_i. We show that they will still all be different after the request. A set S_j only changes, if $r_i \notin S_j$, since only in this case B_j moves a server. Thus, if two sets S_j both contain r_i or both do not contain r_i, they will still be different after r_i.

It remains to treat the case that $r_i \in S_j$ but $r_i \notin S_k$. Since all algorithms served request r_{i-1}, they all have a server on r_{i-1}. Hence, after r_i, S_j will still contain r_{i-1} while S_k will not

(since a server has been moved from r_{i-1} to r_i to serve the request). Hence, even in this case, the sets S_j will remain different.

We have shown that all S_j are different throughout the whole request sequence. This means, that upon request r_i exactly one of the algorithms must move a server at cost $d(r_{i-1}, r_i)$. Thus, the total cost of all B_j combined is

$$\sum_{j=1}^{k} B_j(\sigma) = \sum_{i=2}^{n} d(r_{i-1}, r_i) = \sum_{i=1}^{n-1} d(r_i, r_{i+1}).$$

Up to an additive constant, this is the cost of ALG as shown in (3.6). □

The famous k-server conjecture states that the lower bound from the previous theorem can in fact be obtained:

Conjecture 3.15 (k-Server Conjecture) *For any metric space there is a* k*-competitive algorithm for the* k*-server problem.*

Until today, this conjecture remains unproven. In fact, it is not a priori clear that there should be an algorithm whose competitiver ratio solely depends on k and not on the number of points in X. The first extremely important result was achieved by Fiat et al. [10], who found a $\mathcal{O}((k!)^3)$-competitive algorithm. The most celebrated result in online computation to date is the result of Koutsoupias and Papadimitriou [18] who almost settled the k-server conjecture and showed that the »work function algorithm« achieves a competitive ratio of $(2k-1)$.

While the proof of Koutsoupias and Papadimitriou [18] presents some clever techniques it is simply to long for these lecture notes. Hence, we settle for an easier case, the k-server problem on the real line $\mathbb{R}$ with the Euclidean metric $d(x,y) = |x-y|$.

Algorithm dc (Double Coverage) If request r_i is outside of the convex hull of the servers, then serve the request with the closest server

Otherwise, the request is between two servers. Move both servers at the same speed towards r_i until the first one reaches r_i (if initially two servers occupy the same point, then an arbitrary one is chosen).

Algorithm DC is not a lazy algorithm. However, we can easily make it lazy by the techniques of Lemma 3.13. For the analysis, the non-lazy version given above turns out to be more appropriate.

Theorem 3.16 DC *is* k*-competitive for the* k*-server problem in* $\mathbb{R}$.

Proof: Our proof uses a potential function argument similar to the one for the list accessing problem in Section 2.2. Define M_i to be the minimum weight matching between the servers of DC and OPT after request r_i. We also let S_i be the sum of all distances between the servers of DC at this point in time. Our potential is as follows:

$$\Phi_i := k \cdot M_i + S_i.$$

Clearly, $\Phi_i \geqslant 0$. We imagine that at request r_i first OPT moves and then DC.

Claim 3.17 *(i) If* OPT *moves a server by* δ *units, then the potential increases by at most* $k\delta$.

(ii) If DC *moves two servers by a total amount of δ, then the potential decreases by at least δ.*

Before we prove Claim 3.17 we show how it implies the Theorem. We consider the amortized cost a_i incurred by DC for request r_i, defined by

$$a_i := \mathrm{DC}(r_i) + \Phi_i - \Phi_{i-1}. \tag{3.7}$$

As in Section 2.2 it suffices to show that $a_i \leqslant k \cdot \mathrm{OPT}(r_i)$, since then

$$\mathrm{DC}(\sigma) \leqslant \Phi_0 + \sum_{i=1}^{n} a_i \leqslant \Phi_0 + \sum_{i=1}^{n} k \cdot \mathrm{OPT}(r_i) = k \cdot \mathrm{OPT}(\sigma) + \Phi_0.$$

To see that $a_i \leqslant k \cdot \mathrm{OPT}(r_i)$ note that from Properties 3.17 and 3.17 we have

$$\Phi_i - \Phi_{i-1} \leqslant k \cdot \mathrm{OPT}(r_i) - \mathrm{DC}(r_i).$$

It remains to prove Claim 3.17. If OPT moves a server by δ units, then the matching component of the potential can increase by at most $k\delta$ (just keep the initial minimum cost matching, one of the edge weights increases by δ). Since the sum component S_i remains fixed, this implies 3.17.

To show 3.17 we distinguish between the two different cases which determine how DC reacts to a request. If r_i is outside of the convex hull of DC's servers, then DC just moves one server s, which is either the far most left or the far most right server. Moving s by δ units increases the distance to each of the $k-1$ remaining servers by δ, such that S_i increases by $(k-1)\delta$. On the other hand, it is easy to see that in a minimum cost matching s must be matched to one of OPT's servers who is already at r_i. Hence, moving s closer to its matching partner decreases M_i by at least δ. Hence, we get a net decrease in potential of $(k-1)\delta - k\delta = \delta$ as claimed.

If r_i is between two servers s and s', then both of them will be moved by $\delta/2$, if δ is the total amount of movement by DC in reaction to request r_i. Again, it is straightforward to see that one of the servers s and s' must be matched to OPT's server that is already at r_i. Hence, one server gets $\delta/2$ closer to its partner, while the the second server moves at most $\delta/2$ away from its matching partner. Thus, the matching component of the potential does not increase. Consider now S_i. For any server $s'' \neq s, s'$ the distance $d(s'', s) + d(s'', s')$ remains constant: one term increases by $\delta/2$, the othe decreases by $\delta/2$. However, the distance between s and s' decreases by a total amount of $\delta/2 + \delta/2 = \delta$, so that S_i decreases by δ. Hence, also in the second case Φ must decrease by at least δ. This completes the proof of Claim 3.17. □

3.5 Exercises

Exercise 3.1
Prove Theorem 3.9.

Exercise 3.2
In the coupon collector's problem, there are n types of coupons and at each trial a coupon is chosen at random. Each random coupon is equally likely to be any of the the n types, and the random choice of the coupons are mutually independent. Let X be a random variable defined to be the number of trials required to collect at least one of each type of coupon. Show that $\mathbb{E}[X] = nH_n$.

Hint: Let $C_1, \ldots, C_X$ denote the sequence of trials, where $C_i \in \{1, \ldots, n\}$ denotes the type of coupon drawn in the ith trial. Call the ith trial a *success*, if the type C_i was not drawn

in any of the first $i-1$ selections. Divide the sequence into epochs, where epoch i begins with the trial following the ith success. Consider the random variables X_i ($0 \leqslant i \leqslant n-1$), where X_i denotes the number of trials in the ith epoch.

Exercise 3.3 (Algorithms for the k-server problem)
Show that the online-algorithm which always moves a closest server to the requested point is not competitive for the k-server problem.

Exercise 3.4 (Cow path problem)
A cow is standing in front of a fence with green yummy grassland behind. You understand that the cow is desperately looking for the hole in the fence. Unfortunately, it does not know which way to go: left or right? What would be the best strategy for the hungry cow in order to find the way through the fence as fast as possible?

This problem can be modeled as the search for an unknown point $a \in \mathbb{R}$ starting from the origin 0. The optimal offline strategy knows the place a of the hole in the fence and, thus, the value of the optimal solution (move straight from the origin to a) is simply the distance $|a|$.

(a) Does there exist a strictly c-competitive algorithm?

(b) Assume that $|a| \geqslant 1$. Try to construct an 9-competitive algorithm for that problem.

Hint: Consider an algorithm that moves first $\alpha > 1$ units of length to the right, then goes back to the origin, from where it heads α^2 units of length to the left. The ith turning point of that algorithm is $(-1)^{i+1}\alpha^i$.

What can you say about the competitiveness of that algorithm? How would you choose α?

Exercise 3.5 (Online graph matching)
Consider the following online variant of the *matching Problem*. Given is a bipartite graph $G = (H \cup D, R)$, i.e., each directed edge $r \in R$ is of the form $r = (h, d)$ where $h \in H$ and $d \in D$ and $H \cap D = \varnothing$.

Suppose $G = (V, E)$ is an undirected graph. A *matching* is a subset $M \subseteq E$ such that M contains no two incident edges. In a *maximum matching* it is not possible to add an edge without destroying the matching property. A matching M is *perfect* if each node in V is incident to an edge in M.

The dating service *Online-Matching* received data from n men $H = \{h_1, \ldots, h_n\}$ who are interested in a wonderful lady. The service organizes a dance party where these men can find their love. Therefore invitations has been sent to n promising women $D = \{d_1, \ldots, d_n\}$. With the invitation they received an overview of all interested men. Each lady creates a list of her preferred men. At the entrance to the party each girl is assigned a dance partner from her list. If there is no man from a girls list left without a partner, then she is paid some financial compensation and is sent home. Of course, the goal of the dating service is to partner up as many people as possible.

This problem can be modeled as an *online* version of the problem of finding a perfect matching. We are given a bipartite graph $G = (H \cup D, E)$ with $2n$ nodes, and we assume that G has a perfect matching.

An online algorithm knows all men H from the beginning. A request r_i consists of a neighborhood $N(d_i) = \{h \in H\}(h, d_i) \in E$ of a »woman-node« d_i. The online algorithm has to decide upon the arrival of the request to which man in $N(d_i)$ this girl should be assigned to (if possible). The sequence $\sigma = r_1, \ldots, r_n$ consists of a permutation of ladies and the objective is to obtain as many couples as possible.

Consider, now, the following *very simple* algorithm: as soon as a lady arrives she is assign to *any* still single guy who is on her preference list. If there is no guy left then she is sent home.

(a) Prove that this algorithm is 2-competitive.

(Hint: Assume that M is a maximum matching with $|M| < n/2$. Denote by H' the set of men who got a partner through M. Then, $|H'| < n/2$. Make use of the fact that G has a perfect matching.)

(b) Show that *every* deterministic online algorithm (and even each randomized online algorithm against an adaptive offline adversary) has a competitive ratio of no less than 2 for the problem above.

4 Metrical Task Systems

In this chapter we study a general model for online problems, in which many online problems, such as the list accessing and paging problem, can be captured. Within this framework of **metrical task systems (MTS)**, we introduce an important algorithm: the *work function algorithm*. We refer to [6, Chapter 9] for a more elaborate discussion on this problem.

4.1 Formulation of metrical task systems

In a metrical task system, a server has to process a sequence of tasks that arrive one by one. The server can be in one of a finite number of states and the cost of procesing a task depends on the state of the server. Formally, an MTS is a pair $(\mathcal{M},\mathcal{R})$, where $\mathcal{M}$ is a (finite) metric space and $\mathcal{R}$ is a set of allowable tasks. If there are no restrictions on the tasks, then the corresponding MTS is denoted only by the metric space $\mathcal{M}$. A metric space $\mathcal{M} = (X, d)$ consists of a set of points X and a distance function $d : X \times X \to \mathbb{R}_+$. The distance function, d, is a metric, i.e., it satisfies the following properties:

(i)	$d(x,y) > 0$	$x,y \in X, x \neq y$,
(ii)	$d(x,x) = 0$	$x \in X$,
(iii)	$d(x,y) + d(y,z) \geqslant d(x,z)$	$x,y,z \in X$,
(iv)	$d(x,y) = d(y,x)$	$x,y \in X$.

An example of a metric space is the m-dimensional Euclidean space $(\mathbb{R}^m, \|\cdot\|)$, where $\|\cdot\|$ is the standard euclidean distance. Another example is a metric space induced by a weighted graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}_+$. The set of points is the set of vertices $X = V$ and the distance between two points $u, v \in V$ is the length of the shortest path from u to v. Note that the m-dimensional Euclidean space is an infinite metric space, whereas a graph-induced metric space is finite.

A task τ can be seen as a function $\tau : X \to \mathbb{R}_+ \cup \{\infty\}$. For a finite metric space on $X = \{v_1, \ldots, v_n\}$, a task τ can also be denoted by an n-tuple $< \tau(v_1), \ldots, \tau(v_n) >$.

The value $\tau(x)$, for $x \in X$, denotes the cost of processing task τ in state x. Whenever the server changes state from x to y, it is charged with cost $d(x,y)$. By scaling, we can assume that the minimal distance $\min\{d(x,y) : x,y \in X, x \neq y\} = 1$. The goal in an MTS is, given a sequence of tasks $\sigma = (\tau_1, \ldots, \tau_n)$ and an initial state s_0, to find a sequence of states

$s_1, \ldots, s_l$ minimizing the total cost:

$$\sum_{i=1}^{l} \tau(s_i) + \sum_{i=1}^{l} d(s_{i-1}, s_i),$$

where the first sum denotes the total processing costs and the second is called the total transaction costs.

Example 4.1
Luigi's ice cream problem (cf. Example 1.2) can be formulated as an MTS. The ice cream maker has two states: Vanilla and Chocolate, and the distance between these two states is $d(V, C) = 1$. The set of allowable task is $\mathcal{R} = \{\tau_V, \tau_C\}$. If the machine is in state V making vanilla ice-cream costs 1€ and in state C, the ice-cream needs to be made manually, and it costs 2€. Hence, for a request for making vanilla ice-cream, the corresponding task is $\tau_V =< 1, 2 >$. For chocolate ice-cream, the corresponding task is $\tau_C =< 4, 2 >$. ◁

4.2 A lower bound

In Chapter 2 we have encountered the *cruel adversary* and used it to derive lower bounds on the competitive ratio of any deterministic online algorithm. We now define a family of cruel adversaries for any MTS and show how these adversaries force a competitive ratio of $2n-1$ for any deterministic algorithm on an MTS, $\mathcal{M}$, with n states.

Let $\mathcal{M} = (X, d)$, with $X = \{v_1, \ldots, v_n\}$ be a metric space. An **elementary task** is a task τ with all but one component equal to 0, i.e., $\tau =< 0, \ldots, 0, \tau(v_i), 0, \ldots, 0 >$, where $\tau(v_i) > 0$ is the ith component of τ.

Let us now consider such a deterministic algorithm, ALG. For any $\varepsilon > 0$, we define ADV_ε as the adversary releasing the sequence $\sigma = \tau_1 \ldots \tau_l$, where for each $x \in X$

$$\tau_i(x) = \begin{cases} \varepsilon & \text{if } x = \text{ALG}[i-1], \\ 0 & \text{otherwise.} \end{cases}$$

ADV_ε, thus, releases an elementary task for which ALG has cost ε if it remains in its current state. With respect to ALG we have defined a family $\{\text{ADV}_\varepsilon\}$ of cruel adversaries. Notice that one simple property of these "cruel" task sequences is that the cost of any online algorithm is strictly increasing with the length of the cruel sequence, i.e., $\text{ALG}(\sigma_l) \to \infty$.

To prove a lower bound on the competitive ratio, we must relate the cost of ALG to the optimal offline cost.

Theorem 4.2 *Let* ALG *be a deterministic online algorithm for the* MTS *on* $\mathcal{M} = (X, d)$ *and let* $\varepsilon > 0$. *For each integer* l, *let* $\sigma_l = \tau_1 \ldots \tau_l$ *be the task sequence of length* l *produced by the cruel adversary* ADV_ε *with respect to* ALG. *Then*

$$\limsup_{l \to \infty} \frac{\text{ALG}(\sigma_l)}{\text{OPT}(\sigma_l)} \geqslant \frac{2n-1}{1+2\varepsilon}.$$

Proof: To prove the theorem, we assume that on σ_l ALG changes state k times ($k \leqslant l$), and the states of ALG are $s_0, \ldots, s_k$. The cost of ALG is then

$$\text{ALG}(\sigma_l) = (l-k)\varepsilon + \sum_{1 \leqslant i \leqslant k} d(s_{i-1}, s_i).$$

We, now, need to bound the optimal cost, $\text{OPT}(\sigma_l)$, from above. We do this by using an averaging argument: instead of using OPT, we consider a set, B, of offline (not necessary

optimal) algorithms and we compute the total costs, $B(\sigma_l)$, incurred by these algorithms. At least one of these algorithms has cost at most the average cost, $B(\sigma_l)/|B|$. Hence the optimum has cost at most this average: $\text{OPT}(\sigma_l) \leqslant B(\sigma_l)/|B|$.

The question is: what is the right choice for the set of algorithms? For convenience, let q_0 denote the stat of ALG after processing task τ_i, and we use $q_1, \ldots, q_{n-1}$ to denotethe other states. The set B contains exactly $2n-1$ offline algorithms and preserves the following invariant.

Invariant 1
After processing task τ_i, exactly 2 algorithms occupy state q_j, $j = 1, \ldots, n-1$, and exactly one algorithm occupies state q_0.

This invariant is easily maintained: when ALG does not change its state, all offline algorithms in B remain in their state. Otherwise, when ALG changes its state from q_0 to q_0' exactly one of the two algorithms occupying state q_0' changes its state to q_0.

We can now compute the total cost of these $2n-1$ offline algorithms. For each transition made by ALG, exactly one offline algorithm performs the opposite transition, and there are no other transition costs. Also, exactly one offline algorithm is always in the same state as ALG, so the processing cost, for this, amounts for $(l-k)\varepsilon$. Finally, during those time intervals in which ALG does not pay processing costs ε, i.e., directly after a change of state, two offline algorithms occur processing costs ε. Since there are exactly k such transitions, this amounts to an additional processing cost of $2k\varepsilon$. Hence

$$\begin{aligned} B(\sigma_l) &= (l-k)\varepsilon + \sum_{1 \leqslant i \leqslant k} d(s_i, s_{i-1}) + 2k\varepsilon \\ &= (l-k)\varepsilon + \sum_{1 \leqslant i \leqslant k} d(s_{i-1}, s_i) + 2k\varepsilon. \end{aligned}$$

By averaging, we know that

$$\text{OPT}(\sigma_l) \leqslant \frac{(l-k)\varepsilon + \sum_{1 \leqslant i \leqslant k} d(s_{i-1}, s_i) + 2k\varepsilon}{2n-1} = \frac{\text{ALG}(\sigma_l) + 2k\varepsilon}{2n-1}. \tag{4.1}$$

Since $\text{ALG}(\sigma_l) \geqslant \sum_{1 \leqslant i \leqslant k} d(s_{i-1}, s_i) \geqslant k \min_{x \neq y} d(x, y) = k$, we have that $2k\varepsilon \leqslant 2\varepsilon \text{ALG}(\sigma_l)$. From Equation (4.1), we obtain

$$(2n-1)\text{OPT}(\sigma_l) \leqslant \text{ALG}(\sigma_l)(1+2\varepsilon),$$

from which the theorem follows. □

The desired lower bound of $(2n-1)$ on the competitive ratio of any deterministic algorithm follows immediately from Theorem 4.2: assume there exists an algorithm that is c-competitive, with $c < 2n-1$. By definition, there exists a constant α such that for each task sequence, in particular σ_l, $\text{ALG}(\sigma_l) - c\text{OPT}(\sigma_l) \leqslant \alpha$. This implies that

$$\frac{\text{ALG}(\sigma_l)}{\text{OPT}(\sigma_l)} \leqslant c + \frac{\alpha}{\text{OPT}(\sigma_l)}. \tag{4.2}$$

If $\limsup_{l \to \infty} \text{OPT}(\sigma_l)$ is bounded, then ALG is not competitive at all, as σ_l is constructed so that $\text{ALG}(\sigma_l) \to \infty$. Otherwise, according to Theorem 4.2, there exists sufficiently large l and sufficiently small ε for which Inequality (4.2) is violated. Hence, ALG is not c-competitive with respect to ADV_ε, and $(2n-1)$ is indeed a lower bound.

Corollary 4.3 $(2n-1)$ *is a lower bound on the competitive ratio of any deterministic algorithm for an* MTS *with* n *states.*

Remark 4.4 In the proof of the lower bound, we "cheated" in the sense that we allowed the $2n-1$ offline algorithms to be in their initial state at no cost. That is, the offline algorithms are allowed to choose their initial state. If we would force the offline algorithms to start in the same state s_0 as the online algorithm ALG, then there are to $B(\sigma_l)$ additional transition costs of $2\sum_{i=1}^{n-1} d(s_0, q_i)$, if ALG does not change its state before processing task τ_1, or $2\sum_{i=1}^{n-1} d(s_0, q_i) - 2d(s_0, s_1)$ if ALG changes its state to s_1 before processing this first task.

Note that these additional costs do not depend on the length of the task sequence. Hence, for sufficiently large l and sufficiently small ε the $(2n-1)$ lower bound still holds.

4.3 An optimal work function MTS algorithm

Work functions and work function algorithms play an important role in competitive analysis of deterministic online algorithms. Sometimes, in order to be competitive, an online player needs to keep track of the optimal offline costs so far: this has lead to so-called **work function algorithms** that, in some sense, try to follow an optimal offline solution.

Let $\mathcal{M} = (X, d)$ be an MTS and let s_0 be the initial state. Fix any task sequence $\sigma = \tau_1 \dots \tau_l$. For $i = 1, \dots, l$, $\sigma_i = \tau_1 \dots \tau_i$ denotes the subsequence of the first i tasks in σ. By convention we take $\sigma_0 = \varnothing$, the empty sequence.

For each state $x \in X$, we define $w_i(x)$ to be the minimium (offline) cost to process σ_i, starting from state s_0 and ending in x. Clearly, $\mathrm{OPT}(\sigma) = \min_{x \in X} w_l(x)$. A *dynamic program* that computes $w_l(x)$, and thus $\mathrm{OPT}(\sigma)$ is given by

$$\begin{aligned} w_0(x) &= d(s_0, x), \\ w_i(x) &= \min_{y \in X}\{w_{i-1}(y) + \tau_i(y) + d(y, x)\}. \end{aligned} \tag{4.3}$$

To see that this computes the optimal value, we remark that an optimal sequence to process the tak sequence has the property that no matter what the final transition is, and no matter what the state preceding the final state is, the sequence of transitions form the initial state the the state preceding the final one is optimal.

The functions $w_0, w_1, \dots, w_l$ are called **work functions** of σ with respect to the initial state s_0. The **Work Function Algorithm (wfa)** for an MTS, bases its decisions on these work functions.

Algorithm 4.1 Algorithm WFA

Let s_i be the position of WFA after processing the tasks of σ_i. To process τ_{i+1}, the algorithm moves to a state

$$s_{i+1} = \operatorname*{argmin}_{x \in X}\{w_{i+1}(x) + d(s_i, x)\}, \tag{4.4}$$

with s_{i+1} satisfying

$$w_{i+1}(s_{i+1}) = w_i(s_{i+1}) + \tau_{i+1}(s_{i+1}). \tag{4.5}$$

This definition of WFA raises the question whether there always exists a state s_{i+1} minimizin $w_{i+1}(x) + d(s_i, x)$ and at the same time satisfying Equation (4.5). The following lemma shows that this is always the case.

Lemma 4.5 WFA *is well defined.*

Proof: To show that WFA is well defined, we need to show that there always exists a state s_{i+1} satisfying (4.4) and (4.5). Thereto, we define the set $A = \{y \in X : y = \operatorname{argmin}_{x \in X}\{w_{i+1}(x) + d(s_i, x)\}\}$ of all states satisfying Equation (4.4). Clearly, A is nonempty as X is a nonempty and finite set. We will prove that there exists an element in A satisfying Equation (4.5), implying the correctness of the lemma.

For each state $x \in X$, we know that $w_{i+1}(x) \leqslant w_i(x) + \tau_{i+1}(x)$: the optimal cost of processing σ_{i+1} and ending in state x is not more costly than the optimal way to process σ_i ending in state x and then processing τ_{i+1} without a state change. Adding $d(x, s_i)$ to both sides, we obtain

$$w_{i+1}(x) + d(x, s_i) \leqslant w_i(x) + \tau_{i+1}(x) + d(x, s_i). \tag{4.6}$$

Let $z \in A$ be an arbitrary state minimizing $w_{i+1}(x) + d(s_i, x)$ and let x^* be a preceding state of z if after σ_{i+1}, in the offline setting, we want to finish in z, i.e.,

$$w_{i+1}(z) = w_i(x^*) + \tau_{i+1}(x^*) + d(x^*, z). \tag{4.7}$$

Adding $d(x^*, s_i)$ to both side of Equation (4.7) and rearranging terms yield

$$w_i(x^*) + \tau_{i+1}(x^*) + d(x^*, s_i) = w_{i+1}(z) + d(x^*, s_i) - d(x^*, z).$$

Now, we use the triangle inequality $d(x^*, s_i) \leqslant d(x^*, z) + d(z, s_i)$ in (4.6) and (4.7).

$$w_{i+1}(x^*) + d(s_i, x^*) \leqslant w_i(x^*) + \tau_{i+1}(x^*) + d(x^*, s_i) \leqslant w_{i+1}(z) + d(z, s_i). \tag{4.8}$$

Since $z \in A$, it must be also the case that x^* minimizes $w_{i+1}(x) + d(s_i, x)$, i.e., $x^* \in A$. Moreover, the above inequality holds with equality and so does Inequality (4.6) for $x = x^*$. Hence, $w_{i+1}(x^*) = w_i(x^*) + \tau_{i+1}(x^*)$ and x^* satisfies simultaneously Equations (4.4) and (4.5). □

Now that we have defined WFA and seen that it is well defined, we are ready to prove the competitive ratio of this algorithm. Hereto, we need one more lemma.

Lemma 4.6 *Let s_i be the state of* WFA *after processing τ_i and s_{i+1} the state in which* WFA *processes τ_{i+1}. Then*

$$\begin{aligned} w_{i+1}(s_i) &= w_i(s_{i+1}) + d(s_{i+1}, s_i), \\ w_{i+1}(s_{i+1}) &= w_i(s_{i+1}) + \tau_{i+1}(s_{i+1}). \end{aligned}$$

Proof: The second equation is satisfied by definition of WFA (see also the proof of Lemma 4.5). To prove the first equation, note that for any two states x and y the values of the work function w_{i+1} in these two points do not defer more than $d(x, y)$ from each other. In particular, we have that

$$w_{i+1}(s_i) \leqslant w_{i+1}(s_{i+1}) + d(s_{i+1}, s_i).$$

We, now, need to show that for each $x \in X$

$$w_{i+1}(s_{i+1}) + d(s_{i+1}, s_i) \leqslant w_{i+1}(x) + d(x, s_i).$$

This constraint is also satisfied by definition of WFA (cf. Equation (4.4)). Substituting $x = s_i$ yields the desired equation. □

Theorem 4.7 *Algorithm* WFA *ist $(2n-1)$-competitive for an* MTS *with n states.*

Proof: Let $\sigma = \tau_1 \ldots \tau_l$ be an arbitrary task sequence and let $s_0, s_1, \ldots, s_l$ be the states occupied by WFA on schedule σ: task τ_i is processed in state s_i. Moreover, let $D = \max_{x,y \in X} d(x,y)$. The costs of WFA on this sequence are

$$\begin{aligned}
\text{WFA}(\sigma) &= \sum_{i=1}^{l} (d(s_{i-1}, s_i) + \tau_i(s_i)) \\
&= \sum_{i=0}^{l-1} (d(s_i, s_{i+1}) + \tau_{i+1}(s_{i+1})) \\
&= \sum_{i=0}^{l-1} (w_{i+1}(s_i) - w_i(s_{i+1})), \text{ (by Lemma 4.6)} \\
&= \sum_{i=0}^{l-1} (w_{i+1}(s_i) - w_i(s_i)) \\
&+ \sum_{i=0}^{l-1} (w_{i+1}(s_{i+1}) - w_i(s_{i+1})) \\
&+ w_0(s_0) - w_l(s_l) \\
&\leqslant 2 \sum_{i=0}^{l-1} \max_{x \in X} (w_{i+1}(x) - w_i(x)) - w_l(s_l), \qquad (4.9)
\end{aligned}$$

where the last inequality uses the fact that $w_0(s_0) = d(s_0, s_0) = 0$.

We show, by using a potential function, that

$$\sum_{i=0}^{l-1} \max_{x \in X} (w_{i+1}(x) - w_i(x)) \leqslant n \cdot \text{OPT}(\sigma) + n \cdot D. \qquad (4.10)$$

Exploiting this equation plus the fact that

$$\text{OPT}(\sigma) = \min_{x \in X} w_l(x) = w_l(s^*) \leqslant w_l(s_l),$$

yields from Equation (4.9)

$$\text{WFA}(\sigma) \leqslant (2n-1)\text{OPT}(\sigma) + n \cdot D,$$

from which the theorem follows.

We define the potential function Φ as $\Phi_i = \Phi(w_i) = \sum_{x \in X} w_i(x)$. For this potential function we have

$$\max_{x \in X} (w_{i+1}(x) - w_i(x)) \leqslant \sum_{x \in X} (w_{i+1}(x) - w_i(x)) = \Phi_{i+1} - \Phi_i. \qquad (4.11)$$

Summing (4.11) over $i = 0, \ldots, l-1$, yields:

$$\begin{aligned}
\sum_{i=0}^{l-1} \max_{x \in X} (w_{i+1}(x) - w_i(x)) &\leqslant \Phi_l - \Phi_0 \\
&= \sum_{x \in X} w_l(x) - \sum_{x \in X} w_0(x) \\
&\leqslant \sum_{x \in X} (w_l(s^*) + d(x, s^*)) - \sum_{x \in X} d(s_0, x) \\
&\leqslant n \cdot \text{OPT}(\sigma) + \sum_{x \in X} d(s_0, s^*) \\
&\leqslant n \cdot \text{OPT}(\sigma) + n \cdot D,
\end{aligned}$$

which shows Equation (4.10). □

4.4 Exercises

Exercise 4.1

(a) Formulate the Ski Rental Problem of Example 1.1 as an MTS.

(b) Formulate the paging problem of Chapter 3 as an MTS.

(c) Formulate the list accessing problem described in Chapter 2 as an MTS.

Exercise 4.2

Consider the greedy algorithm for an MTS, that always moves to a state minimizing the transition cost plus the processing cost, i.e., if its current state is s_i, to process task τ_{i+1} it moves to the state s_{i+1} minimizing $d(s_i, x) + \tau_{i+1}(x)$. Show that this algorithm is not competitive at all.

5

Online Scheduling

This chapter is concerned with online scheduling. One of the oldest, or probably *the* oldest, result on competitive analysis was shown in the area of scheduling [13], even though it was back then not presented as an online problem, nor as competitive analysis. In the last 15 years, a lot of attention has been given to online scheduling. See e.g. the survey by Sgall [31] or the PhD thesis of Vestjens [35].

5.1 Scheduling Problems

Scheduling theory is concerned with the *optimal allocation of scarce resources to activities over time*. Scheduling problems arise in a variety of settings.

The models in the area of scheduling are highly standardized: they concern the scheduling of n jobs (the activities) on m machines (the resources), each of which can process no more than one activity at a time, so as to optimize some objective function. The specification of a machine scheduling problem requires the description of a *machine environment*, *job characteristic* and an *optimality criterium*. We use the now-standard notation of [14] for scheduling problems. A problem is denoted by $\alpha|\beta|\gamma$, where α denotes the machine environment, β denotes various side constraints and characteristics, and γ denotes the objective function.

Machine environment. The simplest machine environment is the single machine. For a single machine, the environment is $\alpha = 1$. Other more complex environments involve parallel machines and shops. In the case of parallel machines we are given m machines. A job j with processing requirement p_j can be processed on any of these machines, or, if preeeption is allowed, started on one machine, and when preempted potentially resumed on another machine. A machine can process at most one job at a time and a job can be processed by at most one machine at a time. For the parallel machine environment, there are three standard models: *identical*, *(uniformly) related* and *unrelated* parallel machines.

$\alpha = P$ This denotes the case of *identical parallel machines*. A job j requires p_j units of processing time when processed on any machine.

$\alpha = Q$ In the *uniformly related machines* environment each machine i has a speed $s_i > 0$. A job j, if processed entirely on machine i, would take a total of p_j/s_i time to process.

$\alpha = R$ In the *unrelated parallel machines* w the relative performances of the machines is unrelated. In other words, the speed of machine i on job j, s_{ij} depends on boht the machine and the job; job j requires p_j/s_{ij} time on machine i. We define $p_{ij} := p_j/s_{ij}$.

Open shops, flow shops, and job shops are m-machine environments in which each job consists of several operations, each of which has to be executed on a designated machine; no job can undergo more than one operation at a time. In a *job shop*, the order in which the jobs have to be executed is fixed; in a *fow shop*, this order is fixed and at the same for all jobs; and in an *open shop*, the order is free.

Job characteristics. The job characteristics include the possibility of allowing preemption, and of specifying precedence constraints, release dates, and deadlines. As mentioned above, the parameter β denotes these various side constraints:

- The presence of release dates is indicated by r_j. A job j with release date r_j may not be scheduled before time r_j.
- Deadlines are denoted by d_j. If a job has a deadline, it should be completed before this time.
- If preemption is allowed the entry includes pmtn. In a preemptive schedule a job may be interrupted and resumed later possibly on another machine. In a nonpreemptive schedule, this is not allowed and as soon as a job is started, it has to be processed up to completion.
- Precedence constraints are indicated by prec. A precedence constraint stipulates that a certain job cannot start before another one has completed.

Optimality criteria. A *feasible schedule* is an allocation of the operations or jobs to time intervals on the machine such that all restrictions, such as precedence constraints and release dates are met. The optimality criterium is usually a function of the *completion times* $C_1, \ldots, C_n$ of the jobs. The most common objective functions are

C_{max} the maximum completion time, also called the makespan: $C_{max} = \max_j C_j$;

$\sum w_j C_j$ the average or total (weighted) completion time;

F_{max}, $\sum F_j$ the maximum and average flow time; here the flow time of a job j is defined to be the time difference $C_j - r_j$ between its release time r_j and its completion time C_j.

L_{max} the maximum lateness; for this objective to make sense, one is given deadlines d_j for the jobs and the lateness of j is defined to be $C_j - d_j$.

In this chapter, we mainly focus on single machine scheduling problems to minimize the total (weighted) completion time.

Example 5.1
Since students are usually very short on money[1] you decided to work in Luigi's ice cream shop. Since Example 1.2 Luigi's business has expanded a lot. His ice cream has become famous and he now sells ice cream to many different places all over the world. In particular, his ice cream cakes have become sought after a lot. Luigi own 10 ice cream cake machines to produce the cakes. The machines are identical in the sense that each of them can produce

[1] This mostly holds true for university professors, too.

all of Luigi's recipies at the same cost. Although processing times are the same among the machines, the fancier cakes need more time in the machines than the simpler ones.

It is 08:00 and you arrive a little bit sleepy at Luigi's place. Luigi tells you that there is a total of 280 cakes that need to be produced today. As a big fan of AS Roma Luigi wants to watch the soccer cup final AS Roma vs. Juve at 18:00 and needs to prepare mentally for that big game so that he will have to leave in a few minutes. Luigi is afraid that anyone might steal his famous recipies for the ice cakes (which possibly could be deduced from the running production process) so he wants you to watch the machines until all the ice cakes are finished. »You can assign the cakes to the machines as you want«, he shouts as he slips into his AS Roma soccer shirt and heads out of the shop.

You are left with 280 cake orders and 10 ice cream cake machines. Needless to say that you would also hate to miss the soccer game at 18:00 (although you are a Juve fan and not a supporter of AS Roma). You want to assign jobs to machines so that you can get home as early as possible. In scheduling terms that means you want to *minimize the makespan on idential parallel machines* ($P||C_{max}$). Since the machines are currently being serviced and won't be available until 09:00 you still have some time to think about a schedule that will get you to the soccer game... A reasonable strategy might be to place a cake order on a machine as soon as the machine becomes available. This rule is known as *list scheduling*. A quick calculation shows you that the list schedule will not end before 18:00. However, you know that normally during the day people are coming to order some more (ice cream) cakes. This means that you'll be working until way after 6 p.m. and you will miss the cup final. However, there is one positive thing about the list schedule. You can prove (see Theorem 5.3) that this algorithm will never give a makespan greater than twice the optimal makespan. Thus, the optimal makespan has length at least $(18-9)/2 = 4.5$ hours. Thus, no matter how you schedule the jobs, you won't be able to finish the current demand before 13:30.

This is an example of scheduling jobs on identical parallel machines so as to minimize the makespan. As the cakes of the orders that come in over the day are cannot be started before you know that such an order is made, this problem is also under the restriction of release dates: $P|r_j|C_{max}$. ◁

Example 5.2 (Luigi and the Ice Cream Cakes for the Mafia)
Another day at Luigi's you find yourself in a new situation. Again, there are a couple of cake orders that have to be processed, but this time the situtation has become more complex: Firstly, Luigi is short of some ingredients, and will get deliveries only over the day. This means that for every cake order j, there is a time r_j before which production can not start. Secondly, even the *cosa nostra* has started to order ice cream at Luigi's. Since the *cosa nostra* has higher and lower *mafiosi*, this induces precedences between the cake orders: if the *mafioso* who ordered cake j is ranked higher than the one who ordered k, then production of cake k must not begin before cake j is completed (otherwise Luigi risks ending up in the Tiber with a block of concrete at his feet). Luigi wants to make the *mafiosi* as happy as possible, thus he wants to minimize the average completion time of a cake. Unfortunately, all except for one of his machines are currently being repaired, hence, he needs to process all jobs on just a single ice cream cake machine. His problem is thus $1|r_j, prec|\sum C_j$. ◁

5.2 List Scheduling

Consider the problem of scheduling a sequence $I = (1,\ldots,n)$ of jobs on a collection of m identical machines. Each job has a nonnegative processing requirement p_j and must be processed on a single machine without interruptions. Each machine can run at most one

job at a time (see Figure 5.1). Moreover, you learn about the existence of job j after you have assigned job $j-1$.

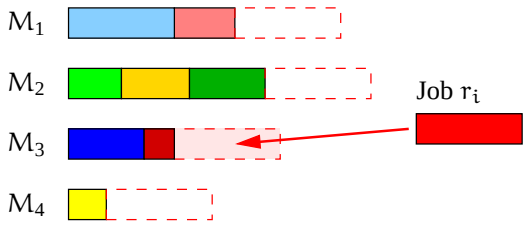

Figure 5.1: Scheduling on 4 identical machines $M_1, \ldots, M_4$. Job j induces the same additional load on each of the machines (indicated by the dashed blocks). In the example, it is assigned to machine M_3.

The goal of the scheduling problem in this section is to minimize the maximum load of a machine, or equivalently, to minimize the length of the schedule. This objective is usually referred to as minimizing the *makespan*.

Let us consider the following *list-scheduling algorithm* that was proposed by Graham in 1966 [13]. This algorithm is also called *Graham's Algorithm*:

Algorithm ls Consider the jobs in the order $1, \ldots, n$. When processing job j assign this job to a machine which currently has the least load.

Observe that LS is in fact an *online algorithm* for the following online scheduling problem: Jobs are given in a request sequence one by one and job j must be scheduled before job $j+1$ becomes known.

Will this algorithm give an optimal solution? Apparently not, if we consider the case of two machines and three jobs with processing times 1, 1, 2. Then, LS will assign the first two jobs to different machines and after the assignment of the third job end up with a makespan of 3. On the other hand, the optimal solution has a makespan of 2.

Although LS does not always yield an optimal solution we can still prove a bound on its performance:

Theorem 5.3 LS *is* $(2-1/m)$*-competitive for the online-problem of scheduling jobs on identical machines such as to minimize the makespan.*

Proof: Set $I = (1, \ldots, n)$ be any sequence of jobs. Without loss of generality we assume that machine 1 is the machine on which the maximum load is achieved by LS. We consider the first time during the processing of I when this load appears on machine 1, say, when job j with processing time p_j is assinged to 1. Let L denote the load of machine 1 prior to the assignment. With these notatations, we have $\text{LS}(I) = L + p_j$.

By construction of the algorithm, at the time when job j is assigned, all machines different from 1 must have load at least L. Hence, the sum of all processing times is at least $mL + p_j$. This gives us the lower bound

$$\text{OPT} \geqslant \frac{mL + p_j}{m} = L + \frac{p_j}{m}.$$

Thus, using $\text{OPT} \geqslant p_j$ we get.

$$\text{LS}(I) = L + p_j \leqslant \text{OPT} + (1 - \frac{1}{m})p_j \leqslant \text{OPT} + (1 - \frac{1}{m})\text{OPT} = (2 - \frac{1}{m})\text{OPT}.$$

This proves the claim. □

While we now interpreteGraham's result as a proof of competitiveness of List Scheduling, it should be stressed that his analysis was deeper. He gives examples where we can increase the makespan by making the problem easier ("timing anomalies" stands for this paradox), namely by either increasing the number of machines, or decreasing the running time of some job, by relaxing precedence constraints, or, finally by reordering the list. He proves that in all of these cases the makespan can change by almost a factor of 2, giving the tight bounds in all cases. The case of reordering the list amounts to the competitive analysis given in Theorem 5.3, as the optimal schedule can be obtained by some particular ordering of the list.

In the follow-up paper [15], Graham shows that the factor of 2 decreases if we modify the algorithm so that some number of long jobs is scheduled first using an optimal schedule, and the rest is scheduled by List Scheduling. Clearly, this algorithm is no longer on-line.

Other two early papers that contain results about online scheduling algorithms are [25, 7]. The first one gives an optimal algorithm for minimizing the makespan of a preemptive schedule on identical machines where jobs arrive over time, and mentions that the algorithm is online. The second paper is, to the best of our knowledge, the first one that states explicitly a lower bound on the performance ratio of any online algorithm for non-preemptive scheduling of jobs with unknown running times on uniform parallel machines.

5.3 Minimizing Average Weighted Completion Time on a Single Machine

The single machine is the easiest machine environment for scheduling, even though many of these problems are already NP-hard. In this section we discuss some elementary results in online scheduling for this machine environment and the objective of minimizing the average or total (weighted) completion time. Some of these results can be used to obtain competitive results for other machine environments.

5.3.1 Preemptive Schedules

The easiest among the problems of minimizing the average weighted completion time on a single machine is the problem in which all weights are equal and preemption is allowed, $1|r_j, pmtn|\sum C_j$. This problem is solved to optimality in polynomial time by the shortest remaining processing time rule.

Shortest Remaining Processing Time (srpt) At each point in time, schedule the job with shortest remaining processing time, preempting when jobs of shorter processing time are released.

Theorem 5.4 ([29]) SRPT *is* 1*-competitive for* $1|r_j, pmtn|\sum C_j$.

Proof: The proof uses an exchange argument. Consider a schedule in which an available job j with the shortest remaining processing time is not being processed at time t, and instead an available job k with strictly larger remaining processing time is run on the machine. We denote by p'_j and p'_k the remaining processing times of j and k at time k. With this notation $p'_j < p'_k$.

After time t, a total of $p'_j + p'_k$ units of time is spent on the processing of j and k. We replace the first p'_j units of time that were allocated to either of jobs j and k after time t and use them to process j until completion. Then, take the remaining p'_k units of time that

were spent processing jobs j and k, and use them to run k. This exchange is possible, since both jobs were available at time t.

In the new schedule, all jobs different from j and k complete at the same time. Job k finishes when job j completed in the original schedule. But job j, which needed $p'_j < p'_k$ additional time, finishes *before* job k originally completed. Thus, the objective function strictly decreases. □

The weighted version of the previous problem, $1|r_j, \text{pmtn}|\sum w_j C_j$, turns out to be NP-hard. Without release dates $1|\text{pmtn}|\sum w_j C_j$ can be solved optimally by Smith's rule [28]: schedule the jobs in non-increasing order of the ratio w_j/p_j. Smith's rule can be interpreted as a weighted version of the shortest processing time rule. Always schedule the job with weighted shortest processing time (weighted by $1/w_j$). Therefore, this rule is also referred to as the weighted shortest processing time rule. A natural extension of this algorithm to the (online) setting with release dates, is the so-called weighted shortest processing time rule (WSPT).

Algorithm wspt At any time t schedule the job with highest ratio w_j/p_j among the available not yet completed jobs. Interrupt the processing of a currently active job, if necessary.

Theorem 5.5 ([30]) WSPT *is* 2*-competitive for the online problem* $1|r_j, \textit{pmtn}|\sum w_j C_j$.

Proof: Consider a job j. Then, the completion time C_j of j in the schedule produced by WSPT is at most r_j plus the sum of the processing times of all jobs k with higher ratio w_k/p_k plus the processing time of j:

$$C_j \leqslant r_j + \sum_{k:\frac{w_k}{p_k} \geqslant \frac{w_j}{p_j}} p_k + p_j. \tag{5.1}$$

Summing over all jobs $j = 1, \ldots, n$ we get from (5.1):

$$\begin{aligned}\sum_{j=1}^{n} w_j C_j &\leqslant \underbrace{\sum_{j=1}^{n} w_j (r_j + p_j)}_{\leqslant \mathrm{OPT}(I)} + \underbrace{\sum_{j=1}^{n} w_j \sum_{k:\frac{w_k}{p_k} \geqslant \frac{w_j}{p_j}} p_k}_{\leqslant \mathrm{OPT}(I)} \\ &\leqslant 2\,\mathrm{OPT}(I).\end{aligned}$$

□

Schulz and Skutella [30] also showed that this analysis is tight, i.e., WSPT cannot be better than 2-competitive. The best known lower bound on the competitive ratio for this problem is 1.073 [9].

Theorem 5.6 ([9]) *Any deterministic preemptive online algorithm for minimizing the total weighted completion time, has a competitive ratio of at least* 1.0730.

Proof: The sequence starts with two jobs arriving at time zero. One job of size $p_1 = 1$ and weight $w_1 = 1$, and the other of size $p_2 = \alpha$ and weight $w_2 = \beta$, where $1 < \beta < \alpha$. Consider an online algorithm ALG at time α. If the smaller job is completed by then, k very small jobs of size $p_j = \varepsilon$ and weight $w_j = \beta\varepsilon$ of total length c arrive ($c = k\varepsilon$). Otherwise, no more jobs arrive. In the first case, OPT runs the larger job, i.e., job 2, then the small jobs

of size $p_j = \varepsilon$, and then the job of unit size, i.e., job 1. For $\varepsilon \to 0$, the cost of OPT on this sequence, I_1, is

$$\text{OPT}(I_1) = \alpha\beta + c\alpha\beta + \beta k(k+1)\varepsilon^2/2 + \alpha + c + 1 = (c+1)(\alpha\beta+1) + \alpha + c^2\beta/2.$$

As ALG may interrupt the processing of a job, we may assume that between time 0 and α ALG was continuously processing a job. That is, ALGis left with a piece of size 1 of job 2. Hence, according to Smith's rule, it does not matter in which order ALG copmpletes the remaining jobs. Its cost is at least

$$\text{ALG}(I_1) \geqslant 1 + \beta(\alpha+1) + (c+1)\alpha\beta + \alpha + c^2\beta/2.$$

In the second case, i.e., ALG has not finished the job of size $p_1 = 1$ by time α and the ε-sized jobs are not released, OPT follows Smith's rule and first schedules the unit-sized job and then the large job. Hence, on this sequence, I_2, the cost for OPT are:

$$\text{OPT}(I_2) = 1 + (\alpha+1)\beta.$$

ALG, on the contrary, can either finish the unit-sized job first, but no earlier than time α, or finishes the larger job first. Hence,

$$\text{ALG}(I_2) \geqslant \min\{\alpha + \beta(\alpha+1), \alpha\beta + \alpha + 1\} = \alpha\beta + \alpha + 1,$$

because $\beta > 1$.

Epstein and Van Stee used a computer to search for good values for α, β, and c, such that the competitive ratio in both cases is high. For $\alpha = 3.4141$, $\beta = 2.5274$, and $c = 4.4580$, the competitive ratio is at least 1.07304. □

These theorems show that there is still a large gap between the performance of WSPT and the best known lower bound. The question was whether the lower bound is weak or that there exists a better algorithm. Recently, Sitters [32] broke the long standing barrier of 2 for the competitive ratio, by designing an online algorithm that is 1.56-competitive. This algorithm is an generalization of WSPT, with a parameter $c > 1$:

Delayed-Preemptive-wspt[c] At any time t schedule the job with highest ratio w_j/p_j among the available not yet completed jobs, with the restriction that a job is never preempted at a moment t if its remaining processing time at time t is at most $(c-1)t$.

Theorem 5.7 ([32]) *Algorithm Delayed-Preemptive-*WSPT*[c] is* 1.56*-competitive for* $c = 1.56$. □

5.3.2 Nonpreemptive Schedules

Up to now, we have considered the preemptive problem, which gives an algorithm more flexibility on its decisions. Now, we will consider non-preemptive scheduling. In non-preemptive scheduling, an online algorithm that has started a job for processing, it needs to complete this job, before it can start working on another one. Therefore, it is intuitively clear that, in this setting, an adversary has more power to fool an online algorithm. Hoogeveen and Vestjens [16] showed that for the unweighted case a significantly higher lower bound than in the preemptive setting.

Theorem 5.8 ([16]) *No online algorithm, for the problem of scheduling jobs non-preemptively on a single machine so as to minimize the total completion time, can achieve a competitive ratio of* $(2-\varepsilon)$*, for any* $\varepsilon > 0$.

Proof: Assume to the contrary that there exists an $(2-\varepsilon)$-competitive algorithm, ALG, for some $\varepsilon > 0$ and consider the following instance. At time $t = 0$, one job of length $p_1 = 1$ is released. ALG needs to start processing this job at a time $S_1 \leqslant 1-\varepsilon$, as otherwise no more jobs are released and the competitive ratio of ALG is $(S_1+1)/1 > 2-\varepsilon$. Then, at time $r_2 = S_1/2 + 1/2 > S_1$, $n-1$ jobs are released with processing time $p_j = 0$. As ALG already started job 1, it cannot start these jobs before time S_1+1, and thus its costs are

$$\text{ALG}(\sigma) \geqslant n(S_1+1).$$

The optimal schedule on the other hand, would delay the starting time of job 1 until time r_2, yielding a value of

$$\text{OPT}(\sigma) = (n-1)r_2 + r_2 + 1 = n(S_1/2+1/2)+1.$$

Hence, for $n \to \infty$, we have that $\text{ALG}(\sigma)/\text{OPT}(\sigma) \to 2$, contradicting the assumption that ALG is $(2-\varepsilon)$-competitive. □

There are several algorithms for this problem that achieve a competitive ratio of 2. Phillips, Stein, and Wein constructed an algorithm based on an optimal preemptive schedule and converted this preemptive schedule in a non-preemptive one by list scheduling [24]. By Theorem 5.4, we know that we can construct an optimal preemptive schedule for the unweighted problem in an online manner.

Algorithm convert-preemptive Given a preemptive schedule P. Let L be an empty list. As soon as a job j has been finished in P, it is added to the list L. At any time t at which the machine is idle, and the list L is non-empty, CONVERT-PREEMPTIVE takes the first job in the list and schedules it.

If the preemptive schedule P is obtained by an online algorithm, CONVERT-PREEMPTIVE itself is also an online algorithm: it (virtually) runs the online algorithm to obtain the preemptive schedule, and as soon as a job is completed, it is added to the list L. We denote by CONVERT-P-SRPT the CONVERT-PREEMPTIVE-algorithm that uses SRPT to obtain the preemptive schedule.

Theorem 5.9 ([24]) *Let C_j^P denote the completion of job j in the preemptive schedule P, and let C_j^N denote its completion time in the non-preemptive schedule obtained by* CONVERT-PREEMPTIVE. *Then $C_j^N \leqslant 2C_j^P$.*

Proof: Consider a job j. At time C_j^P, it is added to the list L and it will be processed as soon as all its predecessors in L have been finished. By definition of CONVERT-PREEMPTIVE, all these jobs have completion time $C_k^P < C_j^P$. Hence, the completion time of job j in the non-preemptive schedule is at most

$$C_j^N \leqslant C_j^P + \sum_{k:C_k^P < C_j^P} p_k + p_j = C_j^P + \sum_{k:C_k^P \leqslant C_j^P} p_k. \tag{5.2}$$

Since all of the jobs in $\{k : C_k^P \leqslant C_j^P\}$ complete at or before C_j^P, the sum of their processing times cannot be larger than C_j^P. Hence, by Equation (5.2), we obtain $C_j^N \leqslant 2C_j^P$. □

Since the objective value of an optimal preemptive schedule is a lower bound for the optimal cost of a non-preemptive schedule, we obtain the following corollary.

Corollary 5.10 *Algorithm* CONVERT-P-SRPT *is 2-competitive.* □

From the original proof of the lower bound by Hoogeveen and Vestjens [16], it is immediately clear that whenever an online algorithm start processing the first job as soon as it gets available, this algorithm can never be competitive: at time $t = 0$ a job of length $p_1 = 1$ arrives and at time $\varepsilon > 0$ $n-1$ jobs of size $p_j = 0$ arrive. Then any algorithm, ALG, that starts processing job 1 at time $t = 0$, will have cost $\text{ALG} = n$, while the optimum first processes the jobs of length $p_j = 0$ and then the job with positive length, and has value $\text{OPT} = n\varepsilon + 1$. By letting $\varepsilon \to 0$, we see that the competitive ratio of such an algorithm is at least n. This observation lead Hoogeveen and Vestjens [16] to design algorithms which introduced unnecessary idle time. The idea of both algorithms was to shift the release date of a job to $\max\{r_j, p_j\}$ resp. $r_j + p_j$ and then apply the shortest processing time rule, which equals Smith's rule for the unweighted case. Recently, Lu et al. [21] generalized this idea to what they called D-SPT. All these algorithms are 2-competitive.

Algorithm d-spt At the release of a job j shift its release date r_j to r'_j, where r'_j is an arbitrary value in the interval $[\max\{r_j, p_j\}, r_j + p_j]$. At any time t at which the machine is idle, schedule the job with shortest processing time that is available according to the shifted release dates.

Theorem 5.11 ([21]) D-SPT *is* 2*-competitive.*

Let σ be the schedule produced by D-SPT on an instance I and let S_j and C_j be, respectively, the starting and completion time of job $j \in \{1, \ldots, n\}$ in this schedule. Based on I and σ, we construct a new instance, $\bar{I}$ as follows. For each job j in I, we define a job also denoted by j, with the same processing time, i.e., $\bar{p}_j = p_j$. The release date of this job is $\bar{r}_j = \min\{S_j, 2r_j + p_j\}$.

Lemma 5.12 *Let* $\text{OPT}(I)$ *and* $\text{OPT}(\bar{I})$ *denote the total completion time of an optimal non-preemptive schedule for, respectively,* I *and* $\bar{I}$*. Then* $\text{OPT}(\bar{I}) \leqslant 2\text{OPT}(I)$.

Proof: Let σ^* be an optimal schedule for I, with completion times C^*_j. We define a schedule $\bar{\sigma}$ by multiplying all completion times by a factor of 2. To be precise, we define the starting time of job j in $\bar{\sigma}$ to be $\bar{S}_j = 2C^*_j - p_j$. It is easy to see that the jobs do not overlap in this schedule. Moreover, since $C^*_j \geqslant 2(r_j + p_j)$, we have $\bar{S}_j \geqslant 2r_j + p_j$, and the schedule is feasible. The total completion time of this schedule is precisely equal to $2\text{OPT}(I)$, and thus $\text{OPT}(\bar{I}) \leqslant 2\text{OPT}(I)$. □

Recall that the Shortest Remaining Processing Time rule yields an optimal schedule for the preemptive problem (Theorem 5.4). This implies that whenever the shortest remaining processing time rule finds a schedule that does not preempt jobs, this schedule is also optimal for the non-preemptive setting. This observation can be used to relate the schedule obtained by D-SPT to the optimal schedule for $\bar{I}$.

Lemma 5.13 *For an instance* I, D-SPT *computes a schedule that is optimal for the instance* $\bar{I}$.

Proof: By the above observation, it suffices to prove that the schedule obtained by D-SPT on instance I coincides with a schedule obtained by SRPT for the instance $\bar{I}$.

First we notice that by definition of the release dates of jobs in $\bar{I}$, no job in the D-SPT-schedule starts before its release date in $\bar{I}$. Moreover, $r'_j \leqslant \bar{r}_j$.

Suppose that at some time t, D-SPT is processing a job j. We have to show that for any other available job, k, in $\bar{I}$ at time t, $p_j(t) \leqslant p_k$, where $p_j(t)$ denotes the remaining processing time of job j at time t.

If $r'_k \leqslant S_j$, then by definition of D-SPT we have that $p_j \leqslant p_k$ and thus $p_j(t) \leqslant p_k$.

If $r'_k > S_j$, then we also have that $r_k + p_k > S_j$. For job j, we know that its starting time is at least $S_j \geqslant p_j$. Hence, its completion time is bounded by $C_j = S_j + p_j \leqslant 2S_j$. The remaining processing time for job j at time t, can now be bounded by $p_j(t) = C_j - t < 2(r_k + p_k) - t = \bar{r}_k + p_k - t$. As job k has been released in the modified instance, we know that $\bar{r}_k \leqslant t$. Hence $p_j(t) < p_k$. □

The proof of Theorem 5.11 now directly follows from Lemmata 5.12 and 5.13.

This proof uses the beautifull technique of relating the value of a schedule obtained by D-SPT to the value of an optimal schedule for a modified instance. To the best of our knowledge, this technique was introduced by Anderson and Potts [1]. These authors considered the weighted version of the non-preemptive scheduling problem on a single machine, $1|r_j|\sum w_j C_j$. Since, the unweighted problem is a special case of the weighted problem (with weights all equal to $w_j = 1$), the lower bound of 2 in Theorem 5.8, is also a lower bound for the weighted problem.

Recall that the problem without release dates can be solved optimally by Smith's rule. Hence, one idea would to extend the idea of [16] to the weighted setting. That is, shift the release dates to $\max\{r_j, p_j\}$ and apply Smith's rule on this modified instance. However, the following example, given by [1], shows that this algorithm cannot be 2-competitive.

Example 5.14
There are two jobs. Job 1 has release date $r_1 = 0$, processing time $p_1 = 3$ and weight $w_1 = 7$. For job 2 these values are $r_2 = 2$, $p_2 = 2$ and $w_2 = 1$. The shifted release dates for these two jobs are $r'_1 = 3$ and $r'_2 = 2$. Hence, the algorithm starts processing job 2 at time $t = 2$ and job 1 at time 4. The total weighted completion time on this instance is $\text{ALG} = 49 + 4 = 53$, whereas the optimum first processes job 1 and then 2, having a value of $\text{OPT} = 21 + 5 = 26$. ◁

The reason that in the above example the competitive ratio of the proposed algorithm is above 2, is that the algorithm started scheduling the job with smallest w_j/p_j, (job 2) even though it could have known a job with higher w_j/p_j is available. To overcome this problem, Anderson and Potts proposed the following algorithm, which they called delayed-WSPT(D-WSPT).

Algorithm d-wspt At any time t at which the machine is available, we choose from among the available jobs, a job j with lowest value of the ratio p_j/w_j. If $p_j \leqslant t$, then we schedule job j to start at time t; otherwise, we do nothing until time p_j or another job is released if this occurs before time p_j.

Theorem 5.15 ([1]) D-WSPT *is* 2*-competitive.* □

5.4 Exercises

Exercise 5.1
Show that any algorithm that does not introduce unnecessary idle time is 1-competitive for the online problem of minimizing the makespan on a single machine, in both the sequence model, i.e., jobs arrive one by one, and the time stamp model.

Exercise 5.2 (ls for uniformly related machines)
The goal of this exercise is to show an upper bound on the competitive of LS for the problem of scheduling on uniformly related machines so as to minimize the makespan.

Recall that in the setting of uniformly related machines each machine has a speed s_i. We assume that $s_1 \geqslant \ldots \geqslant s_m$, and $p_1 \geqslant \ldots \geqslant p_n$.

Algorithm ls for uniformly related machine Consider the jobs in the order $1,\ldots,n$. When processing job j, assign this job to a machine on which it completes earliest.

Notice that in the case that all speeds are equal, i.e., the parallel machine case, this definition of LS coincides with the previously given definition.

(a) Show that the following two lower bounds on the value of an optimal schedule hold:

$$\begin{aligned} \text{OPT} &\geqslant \frac{\sum_j p_j}{\sum_i s_i}, \\ \text{OPT} &\geqslant \frac{p_j}{s_1}. \end{aligned}$$

(b) Let L_i denote the load of machine M_i and let j be the last job assigned to a machine that has maximal load in the LS-schedule. Show that

$$L_i + \frac{p_j}{s_i} \geqslant \text{LS}.$$

(c) Show that

$$\text{LS} \leqslant (1 + \frac{(m-1)s_1}{\sum_i s_i})\text{OPT}.$$

Hint: Rewrite the inequality of exercise (b) and sum appropriately over the machines. Use the second lower bound to bound the value of p_j.

(d) Show that

$$\text{LS} \leqslant \frac{\sum_i s_i}{s_1}\text{OPT}.$$

Hint: Use the inequality in exercise (b) for an appropriate machine.

(e) Show that LS is $(1+\sqrt{4m-3})/2$-competitive for scheduling uniformly related machines.

Hint: Set $x = \frac{s_1}{\sum_i s_i}$ and use the bounds on LS in exercises (c) and (d). For which value of x are these bounds maximal?

Exercise 5.3
Show that the competitive ratio of $\frac{1+\sqrt{5}}{2}$ of LS for two uniformly related machines is the best possible.

Hint: a sequence of 2 jobs suffices.

6 Transportation Problems

6.1 Online Dial-a-Ride Problems

Let $M = (X, d)$ be a metric space with distinguished origin $o \in X$. We assume that M is "connected and smooth´´ in the following sense: for all pairs (x, y) of points from M, there is a rectifiable path $\gamma\colon [0,1] \to X$ in X with $\gamma(0) = x$ and $\gamma(1) = y$ of length $d(x, y)$ (see e.g. [2]). Examples of metric spaces that satisfy the above condition are the Euclidean space $\mathbb{R}^p$ and a metric space induced by an undirected connected edge-weighted graph.

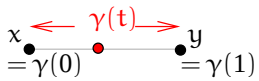

Figure 6.1: Connected and smooth metric space.

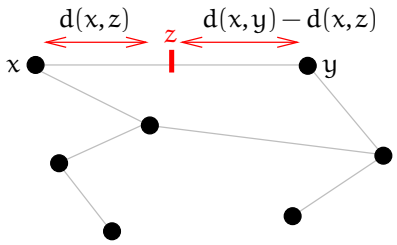

Figure 6.2: A weighted graph induces a connected and smooth metric space.

An instance of the basic online dial-a-ride problem OLDARP in the metric space M consists of a sequence $\sigma = r_1, \ldots, r_n$ of *requests*. ach request is a triple $r_j = (t_j, \alpha_j, \omega_j) \in \mathbb{R} \times X \times X$ with the following meaning: $t_j = t(r_j) \geqslant 0$ is a real number, the time where request r_j is released (becomes known), and $\alpha_j = \alpha(r_j) \in X$ and $\omega_j = \omega(r_j) \in X$ are the source and destination, respectively, between which the object corresponding to request r_j is to be transported.

It is assumed that the sequence $\sigma = r_1, \ldots, r_n$ of requests is given in order of non-decreasing release times, that is, $0 \leqslant t(r_1) \leqslant t(r_2) \leqslant \cdots \leqslant t(r_m)$. For a real number t we denote by $\sigma_{\leqslant t}$ the subsequence of requests in σ released up to and including time t. Similarly, $\sigma_{=t}$ and $\sigma_{<t}$ denote the subsequences of σ consisting of those requests with release time exactly t and strictly smaller than t, respectively.

A server is located at the origin $o \in X$ at time 0 and can move at constant unit speed. We will only consider the case where the server has capacity 1, i.e., it can carry at most one objects at a time. We do not allow *preemption*: once the server has picked up an object, it is not allowed to drop it at any other place than its destination.

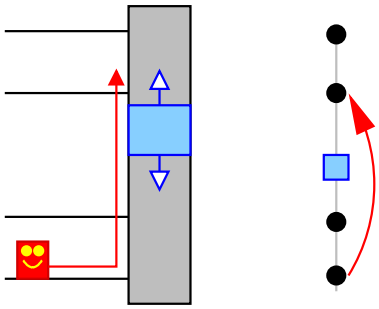

Figure 6.3: Elevator and representation in OLDARP.

An online algorithm for OLDARP does neither have information about the release time of the last request nor about the total number of requests. The online algorithm must determine the behavior of the server at a certain moment t of time as a function of all the requests released up to time t (and the current time t). In contrast, an offline algorithm has information about all requests in the whole sequence σ already at time 0.

A feasible online/offline solution, called *transportation schedule*, for a sequence σ is a sequence of moves for the server such that the following conditions are satisfied: (i) the server starts in the origin at time 0, (ii) each request in σ is served, but picked up not earlier than the time it is released. If additionally (iii) the server ends its work at the origin, then the transportation schedule is called *closed*.Depending on the specific variant of OLDARP only closed schedules may be feasible.

Given an objective function C, the problem C-OLDARP consists of finding a feasible schedule S^* minimizing $C(S^*)$. The problem C-OLDARP can be cast into the framework of request-answer games. We refer to [19] for details. For the purposes in these lecture notes, the more informal definition above will be sufficient.

6.2 Minimizing the Makespan

This section is dedicated to the problem C^o_{max}-OLDARP with the objective to minimize the *makespan*. The makespan $C_{max}(S)$ of a transportation schedule S is defined to be the time when S is completed.

Definition 6.1 (Online Dial-a-Ride Problem C^o_{max}-OLDARP)
The problem C^o_{max}-OLDARP consists of finding a closed transportation schedule which starts at the origin and minimizes the closed makespan C^o_{max}.

The C^o_{max}-OLDARP comprises the *online traveling salesman problem* (OLTSP) which was introduced in [2] as an online variant of the famous traveling salesman problem. In the OLTSP cities (requests) arrive online over time while the salesman is traveling. The requests are to be handled by a salesman-server that starts and ends his work at a designated origin. The cost of such a route is the time when the server has served the last request and has returned to the origin (if the server does not return to the origin at all, then the cost of such a route is defined to be infinity).

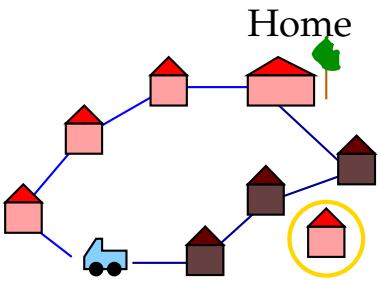

Figure 6.4: Online traveling salesman problem (OLTSP).

Notice that the OLTSP differs from its famous relative, the traveling salesman problem, in certain aspects: First, the cost of a feasible solution is not the length of the tour but the total travel-time needed by the server. The total travel time is obtained from the tour length plus the time where the server remains idle. Second, due to the online nature of the problem it may be unavoidable that a server reaches a certain point in the metric space more than once.

Definition 6.2 (Online Traveling Salesman Problem (OLTSP))
*The **online traveling salesman problem** (OLTSP) is the special case of the C^o_{max}-OLDARP, when for each request r its source and destination coincide, that is, $\alpha(r) = \omega(r)$ for all r.*

If $\sigma = r_1, \ldots, r_n$ is a sequence of requests for the OLTSP we write briefly $r_j = (t_j, \alpha_j)$ instead of $r_j = (t_j, \alpha_j, \alpha_j)$. Observe that the capacity of the server is irrelevant in case of the OLTSP. We will investigate more aspects of the OLTSP in Section 6.4.1.

6.2.1 Lower Bounds

Ausiello et al. established the following lower bounds for the OLTSP:

Theorem 6.3 ([2]) *Any deterministic online algorithm for the OLTSP in general metric spaces has a competitive ratio greater or equal to 2. Any deterministic online algorithm for the OLTSP on the real line has competitive ratio at least $(9+\sqrt{17})/8$.* □

Since C^o_{max}-OLDARP generalizes the OLTSP, the above theorem yields as a corollary a lower bound of 2 on the competitive ratio of any deterministic algorithm for C^o_{max}-OLDARP in general metric spaces.

Theorem 6.4 *Any deterministic online algorithm for the C^o_{max}-OLDARP in general metric spaces has a competitive ratio greater or equal to 2. For the case of the real line, any determinstic algorithm has a competitive ratio at least $c \geqslant 1+\sqrt{2}/2$.*

Proof: As noted above, the general lower bound is an immediate consequence of Theorem 6.3. We now address the case of the real line (with server capacity equal to one). Suppose that ALG is a deterministic online algorithm with competitive ratio $c \leqslant 1+\sqrt{2}/2$. We show that also $c \geqslant 1+\sqrt{2}/2$, which proves the claim of the theorem.

At time 0, the algorithm ALG is faced with two requests $r_1 = (0, o, 2)$ and $r_2 = (0, 2, o)$. The optimal offline cost to serve these two requests is 4.

The server operated by ALG must start serving request r_2 at some time $2 \leqslant T \leqslant 4c-2$, because otherwise ALG could not be c-competitive. At time T the adversary issues another

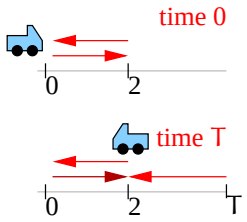

Figure 6.5: Lower bound construction in Theorem 6.4

request $r_3 = (T, T, 2)$. Then $\text{OPT}(r_1, r_2, r_3) = 2T$. On the other hand, $\text{ALG}(r_1, r_2, r_3) \geqslant 3T + 2$. Thus, the competitive ratio c of ALG satisfies

$$c \geqslant \frac{3T+2}{2T} = \frac{3}{2} + \frac{1}{T} \geqslant \frac{3}{2} + \frac{1}{4c-2}.$$

The smallest value $c \geqslant 1$ such that $c \geqslant 3/2 + 1/(4c-2)$ is $c = 1 + \sqrt{2}/2$. This completes the proof. □

We continue with a lower bound for randomized algorithms. The lower bound will be shown for the OLTSP on the real line endowed with the usual Euclidean metric.

Theorem 6.5 *Any randomized algorithm for the* OLTSP *on the real line has competitive ratio greater or equal to* $3/2$ *against an oblivious adversary.*

Proof: We use Yao's Principle (Theorem 2.4) as a tool for deriving the lower bound. No request will be released before time 1. At time 1 with probability $1/2$ there is a request at 1, and with probability $1/2$ a request at -1. This yields a probability distribution X over the two request sequences $\sigma_1 = (1, 1)$ and $\sigma_2 = (1, -1)$.

Since $\text{OPT}(\sigma_1) = \text{OPT}(\sigma_2) = 2$ it follows that $\mathbb{E}_X[\text{OPT}(\sigma_x)] = 2$. We now calculate the expected cost of an arbitrary deterministic algorithm. Consider the deterministic online algorithm ALG_y which has its server at position $y \in \mathbb{R}$ at time 1 (clearly, any deterministic online algorithm is of this form). With probability $1/2$, y is on the same side of the origin as the request which is released at time 1, with probability $1/2$ the position y and the request are on opposite sides of the origin. In the first case, $\text{ALG}_y(\sigma_x) \geqslant 1 + (2 - y)$ (starting at time 1 the server has to move to 1 and back to the origin which needs time at least $2 - y$). In the other case, $\text{ALG}_y(\sigma_x) \geqslant 1 + (2 + y)$. This yields

$$\mathbb{E}_X[\text{ALG}_y(\sigma_x)] = \frac{1}{2}(3 - y) + \frac{1}{2}(3 + y) = 3.$$

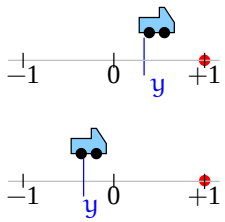

Figure 6.6: Positions of the online server in the randomized lower bound construction in the proof of Theorem 6.5.

Hence $\mathbb{E}_X[\text{ALG}_y(\sigma_x)] \geqslant 3/2 \cdot \mathbb{E}_X[\text{OPT}(\sigma_x)]$ and the claimed lower bound follows by Yao's Principle. □

Corollary 6.6 *Any randomized algorithm for the* C^o_{max}-OLDARP *on the real line has competitive ratio greater or equal to* $3/2$ *against an oblivious adversary.* □

6.2.2 Two Simple Strategies

We now present and analyze two very natural online-strategies for C^o_{max}-OLDARP.

Algorithm 6.1 Algorithm REPLAN

As soon as a new request r_j arrives the server stops and replans: it computes a schedule with minimum length which starts at the current position of the server, takes care of all yet unserved requests (including those that are currently carried by the server), and ends at the origin. Then it continues using the new schedule.

Algorithm 6.2 Algorithm IGNORE

The server remains idle until the point in time t when the first requests become known. The algorithm then serves the requests released at time t immediately, following a shortest schedule S which starts and ends at the origin.

All requests that arrive during the time when the algorithm follows S are temporarily ignored. After S has been completed and the server is back in the origin, the algorithm computes a shortest schedule for all unserved requests and follows this schedule. Again, all new requests that arrive during the time that the server is following the schedule are temporarily ignored. A schedule for the ignored requests is computed as soon as the server has completed its current schedule.

The algorithm keeps on following schedules and temporarily ignoring requests in this way.

Both algorithms above repeatedly solve "offline instances" of the C^o_{max}-OLDARP. These offline instances have the property that all release times are no less than the current time. Thus, the corresponding offline problem is the following: given a number of transportation requests (with release times all zero), find a shortest transportation for them.

For a sequence σ of requests and a point x in the metric space M, let $L^*(t,x,\sigma)$ denote the length of a shortest schedule (i.e., the time difference between its completion time and the start time t) which starts in x at time t, serves all requests from σ (but not earlier than their release times) and ends in the origin.

Observation 6.7 *The function* L^* *has the following properties:*

(i) $L^*(t',x,\sigma) \leqslant L^*(t,x,\sigma)$ *for all* $t' \geqslant t$*;*

(ii) $L^*(t,x,\sigma) \leqslant d(x,y) + L^*(t,y,\sigma)$ *for all* $t \geqslant 0$ *and all* $x,y \in X$*;*

(iii) $\mathrm{OPT}(\sigma) = L^*(0,o,\sigma)$*;*

(iv) $\mathrm{OPT}(\sigma) \geqslant L^*(t,o,\sigma)$ *for any time* $t \geqslant 0$*.*

We now derive another useful property of L^* for the special case that the server has unit-capacity. This result will be used in the proof of the competitiveness of the REPLAN-strategy.

Lemma 6.8 *Let $\sigma = r_1, \ldots, r_n$ be a sequence of requests for the C^o_{max}-OLDARP. Then for any $t \geqslant t_m$ and any request r from σ,*

$$L^*(t, \omega(r), \sigma \backslash \{r\}) \leqslant L^*(t, o, \sigma) - d(\alpha(r), \omega(r)) + d(\alpha(r), o).$$

Here $\sigma \setminus \{r\}$ denotes the sequence obtained from σ by deleting the request r.

Proof: Consider a transportation schedule S^* which starts at the origin o at time t, serves all requests in σ and has length $L^*(t, o, \sigma)$. It suffices to construct another schedule S which starts in $\omega(r)$ no earlier than time t, serves all requests in $\sigma \setminus \{r\}$ and has length at most $L^*(t, o, \sigma) - d(\alpha(r), \omega(r)) + d(\alpha(r), o)$.

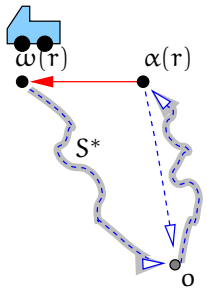

Figure 6.7: Constructing a schedule starting at $\omega(r)$ (dashed lines) from S^* (thick solid lines).

Let S^* serve the requests in the order $r_{j_1}, \ldots, r_{j_m}$ and let $r = r_{j_k}$. Notice that if we start in $\omega(r)$ at time t and serve the requests in the order

$$r_{j_k+1}, \ldots, r_{j_m}, r_{j_1}, \ldots, r_{j_{k-1}}$$

and move back to the origin, we obtain a schedule S with the desired properties. □

Let $\sigma = r_1, \ldots, r_n$ be any request sequence for C^o_{max}-OLDARP. Since the optimal offline algorithm can not serve the last request $r_m = (t_m, \alpha_m, \omega_m)$ from σ before this request is released we get that

$$\text{OPT}(\sigma) \geqslant \max\{L^*(t, o, \sigma), t_m + d(\alpha_m, \omega_m) + d(\omega_m, o)\} \tag{6.1}$$

for any $t \geqslant 0$.

Theorem 6.9 REPLAN *is $5/2$-competitive for the C^o_{max}-OLDARP.*

Proof: Let $\sigma = r_1, \ldots, r_n$ be any sequence of requests. We distinguish between two cases depending on the current load of the REPLAN-server at the time t_m, i.e., the time when the last request is released.

If the server is currently empty it recomputes an optimal schedule which starts at its current position, denoted by $s(t_m)$, serves all unserved requests, and returns to the origin. This schedule has length at most $L^*(t_m, s(t_m), \sigma) \leqslant d(o, s(t_m)) + L^*(t_m, o, \sigma)$. Thus,

$$\begin{aligned} \text{REPLAN}(\sigma) &\leqslant t_m + d(o, s(t_m)) + L^*(t_m, o, \sigma) \\ &\leqslant t_m + d(o, s(t_m)) + \text{OPT}(\sigma) \qquad \text{by (6.1)} \end{aligned} \tag{6.2}$$

Since the REPLAN server has traveled to position $s(t_m)$ at time t_m, there must be a request $r \in \sigma$ where either $d(o, \alpha(r)) \geqslant d(o, s(t_m))$ or $d(o, \omega(r)) \geqslant d(o, s(t_m))$. By the

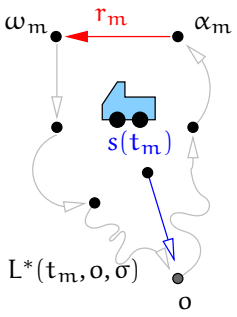

Figure 6.8: Theorem 6.9, Case 1: The server is empty at time t_m when the last request is released.

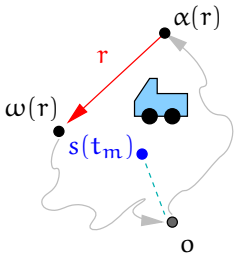

Figure 6.9: $\text{OPT}(\sigma) \geqslant 2\,d(o, s(t_m))$

triangle inequality this implies that the optimal offline server will have to travel at least twice the distance $d(o, s(t_m))$ during its schedule. Thus, $d(o, s(t_m)) \leqslant \text{OPT}(\sigma)/2$. Plugging this result into inequality (6.2) we get that the total time the REPLAN server needs is no more than $5/2\,\text{OPT}(\sigma)$.

It remains the case that the server is serving a request r at the time t_m when the last request r_m becomes known. The time needed to complete the current move is $d(s(t_m), \omega(r))$. A shortest schedule starting at $\omega(r)$ serving all unserved requests has length at most $L^*(t_m, \omega(r), \sigma\backslash\{r\})$. Thus, we have

$$\begin{aligned}
\text{REPLAN}(\sigma) &\leqslant t_m + d(s(t_m), \omega(r)) + L^*(t_m, \omega(r), \sigma\backslash\{r\}) \\
&\leqslant t_m + d(s(t_m), \omega(r)) + L^*(t_m, o, \sigma) \\
&\quad - d(\alpha(r), \omega(r)) + d(\alpha(r), o) \quad \text{by Lemma 6.8} \\
&\leqslant t_m + \text{OPT}(\sigma) - d(\alpha(r), \omega(r)) \\
&\quad + \underbrace{d(s(t_m), \omega(r)) + d(\alpha(r), s(t_m))}_{= d(\alpha(r), \omega(r))} + d(s(t_m), o) \\
&= t_m + d(o, s(t_m)) + \text{OPT}(\sigma).
\end{aligned}$$

Hence, inequality (6.2) also holds in case that the server is carrying an object at time t_m. As argued above, $t_m + d(o, s(t_m)) + \text{OPT}(\sigma) \leqslant 5/2\,\text{OPT}(\sigma)$. This completes the proof. □

Theorem 6.10 *Algorithm* IGNORE *is* $5/2$*-competitive for the* C^o_{max}-OLDARP.

Proof: We consider again the point in time t_m when the last request r_m becomes known. If the IGNORE-server is currently idle at the origin o, then it completes its last schedule

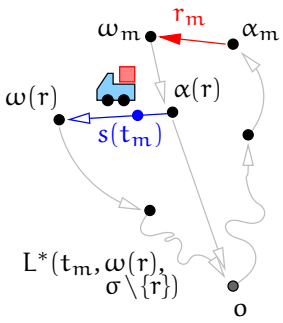

Figure 6.10: Case 2 (C=1): The server is serving request r.

no later than time $t_m + L^*(t_m, o, \sigma_{=t_m})$, where $\sigma_{=t_m}$ is the set of requests released at time t_m.

Since $L^*(t_m, o, \sigma_{=t_m}) \leqslant \text{OPT}(\sigma)$ and $\text{OPT}(\sigma) \geqslant t_m$, it follows that in this case IGNORE completes no later than time $2\,\text{OPT}(\sigma)$.

It remains the case that at time t_m the IGNORE-server is currently working on a schedule S for a subset σ_S of the requests. Let t_S denote the starting time of this schedule. Thus, the IGNORE-server will complete S at time $t_S + L^*(t_S, o, \sigma_S)$. Denote by $\sigma_{\geqslant t_S}$ the set of requests presented after the IGNORE-server started with S at time t_S. Notice that $\sigma_{\geqslant t_S}$ is exactly the set of requests that are served by IGNORE in its last schedule. The IGNORE-server will complete its total service no later than time $t_S + L^*(t_S, o, \sigma_S) + L^*(t_m, o, \sigma_{\geqslant t_S})$.

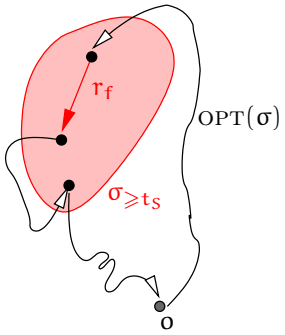

Figure 6.11: Proof of Theorem 6.10: The inequality $\text{OPT}(\sigma) \geqslant t_S + L^*(t_S, \alpha_f, \sigma_{\geqslant t_S})$ holds.

Let $r_f \in \sigma_{\geqslant t_S}$ be the first request from $\sigma_{\geqslant t_S}$ served by OPT. Thus

$$\text{OPT}(\sigma) \geqslant t_f + L^*(t_f, \alpha_f, \sigma_{\geqslant t_S}) \geqslant t_S + L^*(t_m, \alpha_f, \sigma_{\geqslant t_S}). \tag{6.3}$$

Now, $L^*(t_m, o, \sigma_{\geqslant t_S}) \leqslant d(o, \alpha_f) + L^*(t_m, \alpha_f, \sigma_{\geqslant t_S})$ and $L^*(t_S, o, \sigma_S) \leqslant \text{OPT}(\sigma)$. Therefore,

$$\begin{aligned}
\text{IGNORE}(\sigma) &\leqslant t_S + \text{OPT}(\sigma) + d(o, \alpha_f) + L^*(t_m, \alpha_f, \sigma_{\geqslant t_S}) \\
&\leqslant 2\,\text{OPT}(\sigma) + d(o, \alpha_f) && \text{by (6.3)} \\
&\leqslant \frac{5}{2}\,\text{OPT}(\sigma).
\end{aligned}$$

This completes the proof. □

6.2.3 A Best-Possible Online-Algorithm

In this section we present and analyze our algorithm SMARTSTART which achieves a best-possible competitive ratio of 2 (cf. the lower bound given in Theorem 6.3). The idea of the algorithm is basically to emulate the IGNORE-strategy but to make sure that each sub-transportation schedule is completed "not too late": if a sub-schedule would take "too long" to complete then the algorithm waits for a specified amount of time. Intuitively this construction tries to avoid the worst-case situation for IGNORE where right after the algorithm starts a schedule a new request becomes known.

SMARTSTART has a fixed "waiting scaling" parameter $\theta > 1$. From time to time the algorithm consults its "work-or-sleep" routine: this subroutine computes an (approximately) shortest schedule S for all unserved requests, starting and ending in the origin. If this schedule can be completed no later than time θt, i.e., if $t + l(S) \leqslant \theta t$, where t is the current time and $l(S)$ denotes the length of the schedule S, the subroutine returns (S, work), otherwise it returns (S, sleep).

In the sequel it will be convenient to assume that the "work-or-sleep" subroutine uses a ρ-approximation algorithm for computing a schedule: the approximation algorithm always finds a schedule of length at most ρ times the optimal one. While in online computation one is usually not interested in time complexity (and thus in view of competitive analysis we can assume that $\rho = 1$), employing a polynomial-time approximation algorithm will enable us to get a practical algorithm.

The server of algorithm SMARTSTART can assume three states:

idle In this case the server has served all known requests, is sitting in the origin and waiting for new requests to occur.

sleeping In this case the server is sitting at the origin and knows of some unserved requests but also knows that they take too long to serve (what "too long" means will be formalized in the algorithm below).

working In this state the algorithm (or rather the server operated by it) is following a computed schedule.

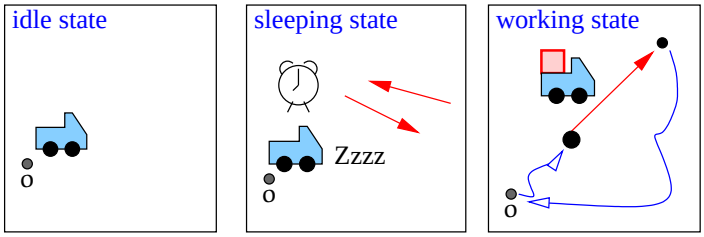

Figure 6.12: States of the SMARTSTART-server.

We now formalize the behavior of the algorithm by specifying how it reacts in each of the three states.

Algorithm 6.3 Algorithm SMARTSTART

If the algorithm is idle at time T and new requests arrive, calls "work-or-sleep". If the result is (S, work), the algorithm enters the working state where it follows schedule S. Otherwise the algorithm enters the sleeping state with wakeup time t', where $t' \geqslant T$ is the earliest time such that $t' + l(S) \leqslant \theta t'$ and $l(S)$ denotes the length of the just computed schedule S, i.e., $t' = \min\{t \geqslant T : t + l(S) \leqslant \theta t\}$.

In the sleeping state the algorithm simply does nothing until its wakeup time t'. At this time the algorithm reconsults the "work-or-sleep" subroutine. If the result is (S, work), then the algorithm enters the working state and follows S. Otherwise the algorithm continues to sleep with new wakeup time $\min\{t \geqslant t' : t + l(S) \leqslant \theta t\}$.

In the working state, i.e, while the server is following a schedule, all new requests are (temporarily) ignored. As soon as the current schedule is completed the server either enters the idle-state (if there are no unserved requests) or it reconsults the "work-or-sleep" subroutine which determines the next state (sleeping or working).

Theorem 6.11 *For all real numbers $\theta \geqslant \rho$ with $\theta > 1$, Algorithm* SMARTSTART *is c-competitive for the C^{o}_{max}-*OLDARP *with*

$$c = \max\left\{\theta, \rho\left(1 + \frac{1}{\theta - 1}\right), \frac{\theta}{2} + \rho\right\}.$$

Moreover, the best possible choice of θ is $\frac{1}{2}\left(1 + \sqrt{1 + 8\rho}\right)$ and yields a competitive ratio of $c(\rho) := \frac{1}{4}\left(4\rho + 1 + \sqrt{1 + 8\rho}\right)$.

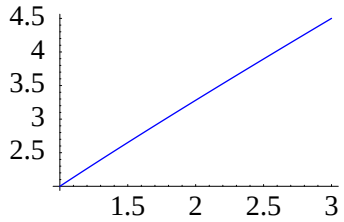

Figure 6.13: Competitive ratio $c(\rho)$ of SMARTSTART for $\rho \geqslant 1$.

Proof: Let $\sigma_{=t_m}$ be the set of requests released at time t_m, where t_m denotes again the point in time when the last requests becomes known. We distinguish between different cases depending on the state of the SMARTSTART-server at time t_m:

Case 1: The server is idle.

In this case the algorithm consults its "work-or-sleep" routine which computes an approximately shortest schedule S for the requests in $\sigma_{=t_m}$. The SMARTSTART-server will start its work at time $t' = \min\{t \geqslant t_m : t + l(S) \leqslant \theta t\}$, where $l(S) \leqslant \rho L^*(t_m, o, \sigma_{=t_m})$ denotes the length of the schedule S.

If $t' = t_m$, then by construction the algorithm completes no later than time $\theta t_m \leqslant \theta\,\mathrm{OPT}(\sigma)$. Otherwise $t' > t_m$ and it follows that $t' + l(S) = \theta t'$. By the performance guarantee ρ of the approximation algorithm employed in "work-or-sleep", we have that

$\text{OPT}(\sigma) \geqslant l(S)/\rho = \frac{\theta-1}{\rho}\, t'$. Thus, it follows that

$$\begin{aligned}\text{SMARTSTART}(\sigma) &= t' + l(S) \\ &\leqslant \theta t' \leqslant \theta \cdot \frac{\rho\,\text{OPT}(\sigma)}{\theta-1} = \rho\left(1+\frac{1}{\theta-1}\right)\text{OPT}(\sigma).\end{aligned}$$

Case 2: The server is sleeping.

Note that the wakeup time of the server is no later than $\min\{t \geqslant t_m : t + l(S) \leqslant \theta t\}$, where S is now a shortest schedule for all the requests in σ not yet served by SMARTSTART at time t_m, and we can proceed as in Case 1.

Case 3: The algorithm is working.

If after completion of the current schedule the server enters the sleeping state, then the arguments given above establish that the completion time of the SMARTSTART-server does not exceed $\rho\left(1+\frac{1}{\theta-1}\right)\text{OPT}(\sigma)$.

The remaining case is that the SMARTSTART-server starts its final schedule S' immediately after having completed S. Let t_S be the time when the server started S and denote by $\sigma_{\geqslant t_S}$ the set of requests presented after the server started S at time t_S. Notice that $\sigma_{\geqslant t_S}$ is exactly the set of requests that are served by SMARTSTART in its last schedule S'.

$$\text{SMARTSTART}(\sigma) = t_S + l(S) + l(S'). \tag{6.4}$$

Here, $l(S)$ and $l(S') \leqslant \rho L^*(t_m, o, \sigma_{\geqslant t_S})$ denotes the length of the schedule S and S', respectively. We have that

$$t_S + l(S) \leqslant \theta t_S, \tag{6.5}$$

since the SMARTSTART only starts a schedule at some time t if it can complete it not later than time θt. Let $r_f \in \sigma_{\geqslant t_S}$ be the first request from $\sigma_{\geqslant t_S}$ served by OPT.

Using the arguments given in the proof of Theorem 6.10 we conclude as in (6.3)that

$$\text{OPT}(\sigma) \geqslant t_S + L^*(t_m, a_f, \sigma_{\geqslant t_S}). \tag{6.6}$$

Moreover, since the tour of length $L^*(t_m, a_f, \sigma_{\geqslant t_S})$ starts in a_f and returns to the origin, it follows from the triangle inequality that

$$L^*(t_m, a_f, \sigma_{\geqslant t_S}) \geqslant d(o, a_f).$$

Thus, from (6.6) we get

$$\text{OPT}(\sigma) \geqslant t_S + d(o, a_f). \tag{6.7}$$

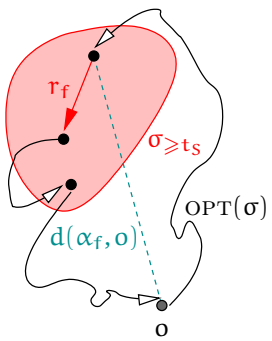

Figure 6.14: The lower bound: $\text{OPT}(\sigma) \geqslant t_S + d(o, a_f)$.

On the other hand

$$\begin{aligned} l(S') &\leqslant \rho\left(d(o,\alpha_f) + L^*(t_m,\alpha_f,\sigma_{\geqslant t_S})\right) \\ &\leqslant \rho\left(d(o,\alpha_f) + \text{OPT}(\sigma) - t_S\right) \qquad \text{by (6.6).} \end{aligned} \tag{6.8}$$

Using (6.5) and (6.8) in (6.4) and the assumption that $\theta \geqslant \rho$, we obtain

$$\begin{aligned} \text{SMARTSTART}(\sigma) &\leqslant \theta t_S + l(S') && \text{by (6.5)} \\ &\leqslant (\theta-\rho)t_S + \rho\, d(o,\alpha_f) + \rho\,\text{OPT}(\sigma) && \text{by (6.8)} \\ &\leqslant \theta\,\text{OPT}(\sigma) + (2\rho-\theta)d(o,\alpha_f) && \text{by (6.7)} \\ &\leqslant \begin{cases} \theta\,\text{OPT}(\sigma) + (2\rho-\theta)\frac{\text{OPT}(\sigma)}{2} & \text{, if } \theta \leqslant 2\rho \\ \theta\,\text{OPT}(\sigma) & \text{, if } \theta > 2\rho \end{cases} \\ &\leqslant \max\left\{\frac{\theta}{2}+\rho, \theta\right\} \text{OPT}(\sigma) \end{aligned}$$

This completes the proof. □

For "pure" competitive analysis we may assume that each schedule S computed by "work-or-sleep" is in fact an optimal schedule, i.e., that $\rho = 1$. The best competitive ratio for SMARTSTART is then achieved for that value of θ where the three terms θ, $1 + \frac{1}{\theta-1}$ and $\frac{\theta}{2}+1$ are equal. This is the case for $\theta = 2$ and yields a competitive ratio of 2. We thus obtain the following corollary.

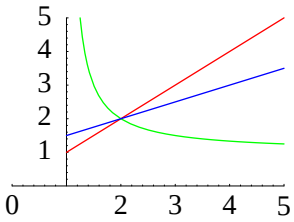

Figure 6.15: The three terms θ, $1 + \frac{1}{\theta-1}$ and $\frac{\theta}{2}+1$ coincide for $\theta = 2$.

Corollary 6.12 *For $\rho = 1$ and $\theta = 2$, Algorithm* SMARTSTART *is 2-competitive for the* C^o_{max}-OLDARP. □

6.3 Minimizing the Sum of Completion Times

In this section se address a different objective function for the for the online dial-a-ride problem, which is motivated by the *traveling repairman problem*, a variant of the traveling salesman problem.

In the *traveling repairman problem* (TRP) a server must visit a set of m points $p_1,\ldots,p_m$ in a metric space. The server starts in a designated point 0 of the metric space, called the *origin*, and travels at most at unit speed. Given a tour through the m points, the completion time C_j of point p_j is defined as the time traveled by the server on the tour until it reaches p_j ($j = 1,\ldots,m$). Each point p_j has a weight w_j, and the objective of the TRP is to find the tour that minimizes the total weighted completion time $\sum_{j=1}^{m} w_j C_j$. This objective is also referred to as the *latency*.

Consider the following online version of the TRP called the *online traveling repairman problem* (OLTRP). Requests for visits to points are released over time while the repairman

(the server) is traveling. In the online setting the completion time of a request r_j at point p_j with release time t_j is the first time at which the repairman visits p_j after the release time t_j. The online model allows the server to wait. However, waiting yields an increase in the completion times of the points still to be served. Decisions are revocable as long as they have not been executed.

In the dial-a-ride generalization $\sum w_j C_j$-OLDARP (for "latency online dial-a-ride problem") each request specifies a ride from one point in the metric space, its *source*, to another point, its *destination*. The server can serve only one ride at a time, and preemption of rides is not allowed: once a ride is started it has to be finished without interruption.

6.3.1 A Deterministic Algorithm

It is not clear how to contruct a competitive algorithm for the $\sum w_j C_j$-OLDARP. Exercise 6.1 asks you to show that the »natural REPLAN-approach« does not work. The approach we present in this section divides time into exponentially increasing intervals and computes transportation schedules for each interval separately. However, each of these subschedules is not choosen such as to minimize the overall objective function. The goal is to maximize the total weight of requests completed within this interval.

Algorithm 6.4 Algorithm INTERVAL$_\alpha$

Phase 0: In this phase the algorithm is initialized.

Set L to be the earliest time when a request could be completed by OPT. We can assume that $L > 0$, since $L = 0$ means that there are requests released at time 0 with source and destination o. These requests are served at no cost. For $i = 0, 1, 2, \ldots$, define $B_i := \alpha^{i-1} L$, where $\alpha \in [1+\sqrt{2}, 3]$ is fixed.

Phase i, for $i = 1, 2, \ldots$: At time B_i compute a transportation schedule S_i for the set of yet unserved requests released up to time B_i with the following properties:

(i) Schedule S_i starts at the endpoint x_{i-1} of schedule S_{i-1} (we set $x_0 := o$).

(ii) Schedule S_i ends at a point x_i with an empty server such that $d(o, x_i) \leqslant B_i$.

(iii) The length of schedule S_i, denoted by $l(S_i)$, satisfies

$$l(S_i) \leqslant \begin{cases} B_1 & \text{if } i = 1 \\ B_i + B_{i-1} & \text{if } i \geqslant 2. \end{cases}$$

(iv) The transportation schedule S_i maximizes the sum of the weights of requests served among all schedules satisfying (i)–(iii).

If $i = 1$, then follow S_1 starting at time B_1. If $i \geqslant 2$, follow S_i starting at time βB_i until βB_{i+1}, where $\beta := \frac{\alpha+1}{\alpha(\alpha-1)}$.

To justify the correctness of the algorithm first notice that $\beta \geqslant 1$ for any $\alpha \geqslant 1+\sqrt{2}$. Moreover, it holds that the transportation schedule S_i computed at time B_i can actually be finished before time βB_{i+1}, the time when transportation schedule S_{i+1}, computed at time B_{i+1}, needs to be started: S_1 is finished latest at time $B_1 + B_1 = 2B_1 \leqslant \frac{\alpha+1}{\alpha-1} B_1 = \beta B_2$ since $\alpha \leqslant 3$. For $i \geqslant 2$, schedule S_i is also finished in time: By condition (iii), $l(S_i) \leqslant B_i + B_{i-1} = (1 + \frac{1}{\alpha}) B_i$. Hence, schedule S_i is finished latest at time $\beta B_i + (1 + \frac{1}{\alpha}) B_i = \frac{\alpha+1}{\alpha-1} B_i = \beta B_{i+1}$.

Lemma 6.13 *Let R_i be the set of requests served by schedule S_i computed at time B_i, $i = 1, 2, \ldots$, and let R_i^* be the set of requests in the optimal offline solution which are*

completed in the time interval $(B_{i-1}, B_i]$. *Then*

$$\sum_{i=1}^{k} w(R_i) \geqslant \sum_{i=1}^{k} w(R_i^*) \quad \text{for } k = 1, 2, \ldots .$$

Proof: We first argue that for any $k \geqslant 1$ we can obtain from the optimal offline solution S^* a schedule S which starts in the origin, has length at most B_k, ends with an empty server at a point with distance at most B_k from the origin, and which serves all requests in $\bigcup_{i=1}^{k} R_i^*$.

Consider the optimal offline transportation schedule S^*. Start at the origin and follow S^* for the first B_k time units with the modification that, if a request is picked up in S^* before time B_k but not delivered before time B_k, omit this action. Observe that this implies that the server is empty at the end of this schedule. We thereby obtain a schedule S of length at most B_k which serves all requests in $\bigcup_{i=1}^{k} R_i^*$. Since the server moves at unit speed, it follows that S ends at a point with distance at most B_k from the origin.

We now consider phase k and show that by the end of phase k, at least requests of weight $\sum_{i=1}^{k} w(R_i^*)$ have been scheduled by INTERVAL$_\alpha$. If $k = 1$, the transportation schedule S obtained as outlined above satisfies already all conditions (i)–(iii) required by INTERVAL$_\alpha$. If $k \geqslant 2$, then condition (i) might be violated, since S starts in the origin. However, we can obtain a new schedule S' from S starting at the endpoint x_{k-1} of the schedule from the previous phase, moving the empty server from x_{k-1} to the origin and then following S. Since $d(x_{k-1}, o) \leqslant B_{k-1}$, the new schedule S' has length at most $B_{k-1} + l(S) \leqslant B_{k-1} + B_k$ which means that it satisfies all the properties (i)–(iii) required by INTERVAL$_\alpha$.

Recall that schedule S and thus also S' serves all requests in $\bigcup_{i=1}^{k} R_i^*$. Possibly, some of the requests from $\bigcup_{i=1}^{k} R_i^*$ have already been served by INTERVAL$_\alpha$ in previous phases. As omitting requests can never increase the length of a transportation schedule, in phase k, INTERVAL$_\alpha$ can schedule at least all requests from

$$\left(\bigcup_{i=1}^{k} R_i^* \right) \setminus \left(\bigcup_{i=1}^{k-1} R_i \right).$$

Consequently, the weight of all requests served in schedules $S_1, \ldots, S_k$ of INTERVAL$_\alpha$ is at least $w(\bigcup_{i=1}^{k} R_k^*) = \sum_{i=1}^{k} w(R_k^*)$ as claimed. □ □

The previous lemma gives us the following bound on the number of phases that INTERVAL$_\alpha$ uses to process a given input sequence σ.

Corollary 6.14 *Suppose that the optimum offline schedule is completed in the interval* $(B_{p-1}, B_p]$ *for some* $p \geqslant 1$. *Then the number of phases of the Algorithm* INTERVAL$_\alpha$ *is at most* p. *Schedule* S_p *computed at time* B_p *by* INTERVAL$_\alpha$ *is completed no later than time* βB_{p+1}.

Proof: By Lemma 6.13 the weight of all requests scheduled in the first p phases equals the total weight of all requests. Hence all requests must be scheduled within the first p phases. Since, by construction of INTERVAL$_\alpha$, schedule S_p computed in phase p completes by time βB_{p+1}, the claim follows. □ □

To prove competitiveness of INTERVAL$_\alpha$ we need an elementary lemma which can be proven by induction.

Lemma 6.15 *Let* $a_i, b_i \in \mathbb{R}_{\geqslant 0}$ *for* $i = 1, \ldots, p$, *for which*

(i) $\sum_{i=1}^{p} a_i = \sum_{i=1}^{p} b_i$;

(ii) $\sum_{i=1}^{p'} a_i \geqslant \sum_{i=1}^{p'} b_i$ *for all* $1 \leqslant p' \leqslant p$.

Then $\sum_{i=1}^{p} \tau_i a_i \leqslant \sum_{i=1}^{p} \tau_i b_i$ *for any nondecreasing sequence* $0 \leqslant \tau_1 \leqslant \cdots \leqslant \tau_p$. □

Proof: See Exercise 6.2. □

Theorem 6.16 *Algorithm* INTERVAL$_\alpha$ *is* $\frac{\alpha(\alpha+1)}{\alpha-1}$*-competitive for the* $\sum w_j C_j$-OLDARP *for any* $\alpha \in [1+\sqrt{2}, 3]$. *For* $\alpha = 1+\sqrt{2}$*, this yields a competitive ratio of* $(1+\sqrt{2})^2 < 5.8285$.

Proof: Let $\sigma = r_1, \ldots, r_n$ be any sequence of requests. By definition of INTERVAL$_\alpha$, each request served in schedule S_i completes no later than time $\beta B_{i+1} = \frac{\alpha+1}{\alpha(\alpha-1)} B_{i+1}$. Summing over all phases $1, \ldots, p$ yields

$$\text{INTERVAL}_\alpha(\sigma) \leqslant \frac{\alpha+1}{\alpha(\alpha-1)} \sum_{i=1}^{p} B_{i+1} w(R_i) = \alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{i=1}^{p} B_{i-1} w(R_i). \tag{6.9}$$

From Lemma 6.13 we know that $\sum_{i=1}^{k} w(R_i) \geqslant \sum_{i=1}^{k} w(R_i^*)$ for $k = 1, 2, \ldots$, and from Corollary 6.14 we know that $\sum_{i=1}^{p} w(R_i) = \sum_{i=1}^{p} w(R_i^*)$. Therefore, application of Lemma 6.15 to the sequences $a_i := w(R_i)$ and $b_i := w(R_i^*)$ with the weighing sequence $\tau_i := B_{i-1}$, $i = 1, \ldots, p$, gives

$$\alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{i=1}^{p} B_{i-1} w(R_i) \leqslant \alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{i=1}^{p} B_{i-1} w(R_i^*). \tag{6.10}$$

Denote by C_j^* the completion time of request r_j in the optimal offline solution OPT(σ). For each request r_j denote by $(B_{\phi_j}, B_{\phi_{j+1}}]$ the interval that contains C_j^*. Then

$$\alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{i=1}^{p} B_{i-1} w(R_i^*) = \alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{j=1}^{m} B_{\phi_j} w_j \leqslant \alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{j=1}^{m} w_j C_j^*. \tag{6.11}$$

(6.9), (6.10), and (6.11) together yield

$$\text{INTERVAL}_\alpha(\sigma) \leqslant \alpha \cdot \frac{\alpha+1}{\alpha-1} \cdot \text{OPT}(\sigma).$$

The value $\alpha = 1+\sqrt{2}$ minimizes the function $f(\alpha) := \frac{\alpha(\alpha+1)}{\alpha-1}$ in the interval $[1+\sqrt{2}, 3]$, yielding $(1+\sqrt{2})^2 < 5.8285$ as competitive ratio. □ □

Corollary 6.17 *For* $\alpha = 1+\sqrt{2}$*, algorithm* INTERVAL$_\alpha$ *is* $(1+\sqrt{2})^2$*-competitive for the* OLTRP. □

6.3.2 An Improved Randomized Algorithm

In this section we use randomization to improve the competitiveness result obtained in the previos section. At the beginning, RANDINTERVAL$_\alpha$ chooses a random number $\delta \in (0, 1]$ according to the uniform distribution. From this moment on, the algorithm is completely deterministic, working in the same way as the deterministic algorithm INTERVAL$_\alpha$ presented in the previous section. For $i \geqslant 0$ define $B_i' := \alpha^{i-1+\delta} L$, where again L is the earliest time that a request could be completed by OPT. As stated before in the case of INTERVAL$_\alpha$ we can assume that $L > 0$.

The difference between RANDINTERVAL$_\alpha$ and INTERVAL$_\alpha$ is that all phases are defined using $B'_i := \alpha^{i-1+\delta}L$ instead of $B_i := \alpha^{i-1}L$, $i \geqslant 1$. To justify the accuracy of RANDINTERVAL$_\alpha$, note that in the correctness proof for the deterministic version, we only made use of the fact that $B_{i+1} = \alpha B_i$ for $i \geqslant 0$. This also holds for the B'_i. Hence, any choice of the parameter $\alpha \in [1+\sqrt{2}, 3]$ yields a correct version of RANDINTERVAL$_\alpha$. We will show later that the optimal choice for RANDINTERVAL$_\alpha$ is $\alpha = 3$.

The proof of Lemma 6.13 also holds also with B_i replaced by B'_i for each $i \geqslant 0$. We thus obtain the following lemma.

Lemma 6.18 *Let R_i be the set of requests scheduled in phase $i \geqslant 1$ of Algorithm* RANDINTERVAL$_\alpha$ *and denote by R^*_i the set of requests that are completed by* OPT *in the time interval $(B'_{i-1}, B'_i]$. Then*

$$\sum_{i=1}^{k} w(R_i) \geqslant \sum_{i=1}^{k} w(R^*_i) \quad \textit{for } k = 1, 2, \ldots .$$

Proof: We only have to ensure that schedule S_1 is finished before time $\beta B'_2$. The rest of the proof is the same as that for Lemma 6.13. The proof for the first phase follows from the fact that $\beta B'_2 - L = (\frac{\alpha+1}{\alpha(\alpha-1)} \cdot \alpha^{1+\delta} - 1)L > \alpha^{+\delta}L = B'_1$. □

We can now use the proof of Theorem 6.16 with Lemma 6.13 replaced by Lemma 6.18. This enables us to conclude that for a sequence $\sigma = r_1, \ldots, r_n$ of requests the expected objective function value of RANDINTERVAL$_\alpha$ satisfies

$$\begin{aligned} \mathbb{E}\left[\text{RANDINTERVAL}_\alpha(\sigma)\right] &\leqslant \mathbb{E}\left[\alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{j=1}^{m} B'_{\phi_j} w_j\right] \\ &= \alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{j=1}^{m} w_j \mathbb{E}\left[B'_{\phi_j}\right], \end{aligned} \tag{6.12}$$

where $(B'_{\phi_j}, B'_{\phi_{j+1}}]$ is the interval containing the completion time C^*_j of request r_j in the optimal solution OPT(σ).

To prove a bound on the performance of RANDINTERVAL$_\alpha$ we compute $\mathbb{E}\left[B'_{\phi_j}\right]$. Notice that B'_{ϕ_j} is the largest value $\alpha^{k+\delta}L$, $k \in \mathbb{Z}$, which is strictly smaller than C^*_j.

Lemma 6.19 *Let $z \geqslant L$ and $\delta \in (0,1]$ be a random variable uniformly distributed on $(0,1]$. Define B by $B := \max\{\alpha^{k+\delta}L : \alpha^{k+\delta}L < z$ and $k \in \mathbb{Z}\}$. Then, $\mathbb{E}[B] = \frac{\alpha-1}{\alpha \ln \alpha} \cdot z$.*

Proof: Suppose that $\alpha^k L \leqslant z < \alpha^{k+1}L$ for some $k \geqslant 0$. Observe that

$$B = \begin{cases} \alpha^{k-1+\delta}L & \text{if } \delta \geqslant \log_\alpha \frac{z}{\alpha^k L} \\ \alpha^{k+\delta}L & \text{otherwise.} \end{cases}$$

Hence

$$\begin{aligned} \mathbb{E}[B] &= \int_0^{\log_\alpha \frac{z}{\alpha^k L}} \alpha^{k+\delta} L \, d\delta + \int_{\log_\alpha \frac{z}{\alpha^k L}}^{1} \alpha^{k-1+\delta} L \, d\delta \\ &= \alpha^k L \left[\frac{1}{\ln \alpha} \alpha^\delta\right]_0^{\log_\alpha \frac{z}{\alpha^k L}} + \alpha^{k-1} L \left[\frac{1}{\ln \alpha} \alpha^\delta\right]_{\log_\alpha \frac{z}{\alpha^k L}}^{1} = \frac{\alpha-1}{\alpha \ln \alpha} \cdot z. \end{aligned}$$

This completes the proof. □

From Lemma 6.19 we can conclude that $\mathbb{E}\left[B'_{\phi_j}\right] = \frac{\alpha-1}{\alpha \ln \alpha} \cdot C^*_j$. Using this result in inequality (6.12), we obtain

$$\mathbb{E}\left[\text{RANDINTERVAL}_\alpha(\sigma)\right] \leqslant \alpha \cdot \frac{\alpha+1}{\alpha-1} \cdot \frac{\alpha-1}{\alpha \ln \alpha} \cdot \sum_{j=1}^{m} w_j C^*_j = \frac{\alpha+1}{\ln \alpha} \cdot \text{OPT}(\sigma).$$

Minimizing the function $g(\alpha) := \frac{\alpha+1}{\ln \alpha}$ over the interval $[1+\sqrt{2}, 3]$, we conclude that the best choice is $\alpha = 3$.

Theorem 6.20 *Algorithm* RANDINTERVAL$_\alpha$ *is* $\frac{\alpha+1}{\ln \alpha}$*-competitive for the* $\sum w_j C_j$*-*OLDARP *against an oblivious adversary, where* $\alpha \in [1+\sqrt{2}, 3]$*. Choosing* $\alpha = 3$ *yields a competitive ratio of* $\frac{4}{\ln 3} < 3.6410$ *for* RANDINTERVAL$_\alpha$ *against an oblivious adversary.* □

Corollary 6.21 *For* $\alpha = 3$*, algorithm* RANDINTERVAL$_\alpha$ *is* $\frac{4}{\ln 3}$*-competitive for the* OLTRP *against an oblivious adversary.* □

6.4 Alternative Adversary Models

6.4.1 The Fair Adversary

We consider the OLTSP on $\mathbb{R}_{\geqslant 0}$ in case that the offline adversary is the conventional (omnipotent) opponent.

Theorem 6.22 *Let* ALG *be any deterministic algorithm for* OLTSP *on* $\mathbb{R}_{\geqslant 0}$*. Then the competitive ratio of* ALG *is at least* $3/2$*.*

Proof: At time 0 the request $r_1 = (0,1)$ is released. Let $T \geqslant 1$ be the time that the server operated by ALG has served the request r_1 and returned to the origin o. If $T \geqslant 3$, then no further request is released and ALG is no better than $3/2$-competitive since $\text{OPT}(r_1) = 2$. Thus, assume that $T < 3$.

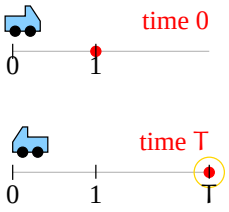

Figure 6.16: Lower bound construction of Theorem 6.22.

In this case the adversary releases a new request $r_2 = (T, T)$. Clearly, $\text{OPT}(r_1, r_2) = 2T$. On the other hand $\text{ALG}(r_1, r_2) \geqslant 3T$, yielding a competitive ratio of $3/2$. □

The following simple strategy achieves a competitive ratio that matches this lower bound (as we will show below):

Algorithm 6.5 Algorithm MRIN ("Move-Right-If-Necessary")

If a new request is released and the request is to the right of the current position of the server operated by MRIN, then the MRIN-server starts to move right at unit speed. The server continues to move right as long as there are yet unserved requests to the right of the server. If there are no more unserved requests to the right, then the server moves towards the origin o at unit speed.

Theorem 6.23 MRIN *is a* $3/2$*-competitive algorithm for the* OLTSP *on* $\mathbb{R}_{\geqslant 0}$.

Proof: See Exercise 6.3. □

The adversary used in Theorem 6.22 abused his power in the sense that he moves to points where he knows a request will pop up without revealing the request to the online server before reaching the point.

The model of the *fair adversary* defined formally below allows improved competitive ratios for the OLTSP on $\mathbb{R}_0^+$. Under Recall that $\sigma_{<t}$ is the subsequence of σ consisting of those requests with release time strictly smaller than t.

Definition 6.24 (Fair Adversary) *An offline adversary for the* OLTSP *in the Euclidean space* $(\mathbb{R}^n, \|.\|)$ *is* ***fair****, if at any moment* t*, the position of the server operated by the adversary is within the convex hull of the origin* o *and the requested points from* $\sigma_{<t}$.

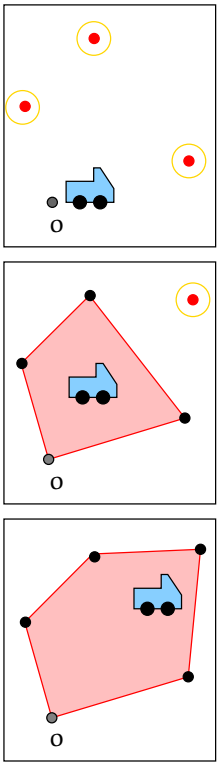

Figure 6.17: The fair adversary moves in the convex hull of points released.

In the special case of $\mathbb{R}_{\geqslant 0}$ a fair adversary must always keep its server in the interval $[0, F]$, where F is the position of the request with the largest distance to the origin $o = 0$ among all requests released so far.

The following lower bound results show that the OLTSP against a fair adversary is still a non-trivial problem.

Theorem 6.25 *Let* ALG *be any deterministic algorithm for* OLTSP *on* $\mathbb{R}$. *Then the competitive ratio of* ALG *against a fair adversary is at least* $(5+\sqrt{57})/8$.

Proof: See [19, 5]. □

Theorem 6.26 *Let* ALG *be any deterministic algorithm for* OLTSP *in* $\mathbb{R}_{\geqslant 0}$. *Then the competitive ratio of* ALG *against a fair adversary is at least* $(1+\sqrt{17})/4$.

Proof: See [19, 5]. □

We now show that MRIN performs better against the fair adversary.

Theorem 6.27 *Algorithm* MRIN*is* $4/3$*-competitive for the* OLTSP *on* $\mathbb{R}_{\geqslant 0}$ *against a fair adversary.*

Proof: We use induction on the number of requests in the sequence σ to establish the claim. The claim clearly holds if σ contains at most one request. The induction hypothesis states that the claim of the theorem holds for any sequence of $m-1$ requests.

Let $\sigma = r_1, \ldots, r_n$ be any sequence of requests. We consider the time $t := t_m$ when the last set of requests $\sigma_{=t_m}$ is released. If $t = 0$, then the claim obviously holds, so we will assume for the remainder of the proof that $t > 0$. Let $r = (t, x)$ be that request of $\sigma_{=t_m}$ which is furthest away from the origin.

In the sequel we denote by $s(t)$ and $s^*(t)$ the positions of the MRIN-server and the fair adversary server at time t, respectively.

Let $r_f = (t_f, f)$ be the furthest unserved request by MRIN of the subsequence $\sigma_{<t}$ at time t, that is, the unserved request from $\sigma_{<t}$ most remote from the origin o. Finally, let $r_F = (t_F, F)$ be the furthest request in $\sigma_{<t}$. Notice that by definition $f \leqslant F$.

We distinguish three different cases depending on the position x of the request r relative to f and F.

Case 1: $x \leqslant f$
Since the MRIN-server still has to travel to f, all the requests in $\sigma_{=t}$ will be served on the way back to the origin and the total completion time of the MRIN-server will not increase by releasing the requests $\sigma_{=t}$. Since new requests can never decrease the optimal offline solution value, the claim follows from the induction hypothesis.

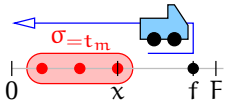

Figure 6.18: Case 1: If $x \leqslant f$, MRIN's cost does not increase.

Case 2: $f \leqslant x < F$
If $s(t) \geqslant x$, again MRIN's completion time does not increase compared to the situation before the requests in $\sigma_{=t}$ were released, so we may assume that $s(t) \leqslant x$.

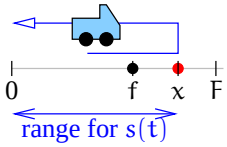

Figure 6.19: Case 2: $f \leqslant x < F$ and $s(t) \leqslant x$.

The MRIN-server will now travel to point x which needs time $d(s(t), x)$, and then return to the origin. Thus, $\text{MRIN}(\sigma) = t + d(s(t), x) + x$. On the other hand $\text{OPT}(\sigma) \geqslant t + x$. It follows that

$$\frac{\text{MRIN}(\sigma)}{\text{OPT}(\sigma)} \leqslant 1 + \frac{d(s(t), x)}{\text{OPT}(\sigma)} \tag{6.13}$$

We now show that $\text{OPT}(\sigma)$ is at least 3 times $d(s(t),x)$, this will establish the claimed ratio of $4/3$. Notice that $f < F$ (since $f \leqslant x < F$) and the fact that f is the furthest unserved request at time t implies that the MRIN-server must have already visited F at time t (otherwise the furthest unserved request would be at F and not at $f < F$). Therefore, $t \geqslant F + d(F, s(t))$, and

$$\text{OPT}(\sigma) \geqslant t + x \geqslant F + d(F, s(t)) + x. \tag{6.14}$$

Clearly, each of the terms on the right hand side of inequality (6.14) is at least $d(s(t),x)$.

Case 3: $f \leqslant F \leqslant x$
First recall that the MRIN-server always moves to the right if there are yet unserved requests to the right of his present position.

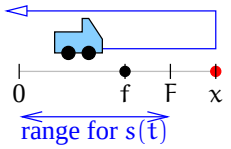

Figure 6.20: Case 3: $f \leqslant F \leqslant x$.

Since the last request (t,x) is at least as far away from the origin as F, the optimal offline server will only move left after it has served the furthest request in σ, in this case at x. In fact, the optimal fair offline strategy is as follows: as long as there are unserved requests to the right of the server, move right, otherwise wait at the current position. As soon as the last request (t,x) has been released and the offline server has reached x, it moves to the origin and completes its work.

Hence, at any time in the interval $[0,t]$, the fair adversary's server is to the right of the MRIN-server or at the same position.

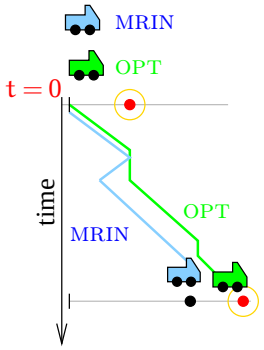

Figure 6.21: The fair adversary is always to the right of the MRIN-server.

Because the offline server does not move left as long as there will be new requests released to the right of its current position, the distance between the MRIN-server and the offline server increases only if the offline server is waiting at some point. Let $W^*(t)$ be the total waiting time of the offline server at the moment t when the last request x is released. Then we know that

$$d(s(t), s^*(t)) \leqslant W^*(t). \tag{6.15}$$

Moreover, the following relation between the current time and the waiting time holds:

$$t = d(o, s^*(t)) + W^*(t). \tag{6.16}$$

Since the adversary is fair, its position $s^*(t)$ at time t can not be to the right of F. Thus, $d(s^*(t),x) = d(s^*(t),F) + d(F,x)$ which gives us

$$\begin{aligned}
\text{OPT}(\sigma) &\geqslant t + d(s^*(t),F) + d(F,x) + x && (6.17)\\
&= d(o,s^*(t)) + W^*(t) + d(s^*(t),F) + d(F,x) + x && \text{by (6.16)}\\
&= W^*(t) + F + d(F,x) + x\\
&= W^*(t) + 2x\\
&\geqslant W^*(t) + 2d(s(t),s^*(t))\\
&\geqslant 3d(s(t),s^*(t)) && \text{by (6.15)} \quad (6.18)
\end{aligned}$$

At time t MRIN's server has to move from its current position $s(t)$ to x and from there to move to the origin:

$$\begin{aligned}
\text{MRIN}(\sigma) &= t + d(s(t),x) + x\\
&= t + d(s(t),s^*(t)) + d(s^*(t),F) + d(F,x) + x.
\end{aligned}$$

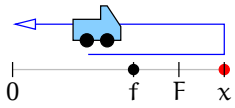

Figure 6.22: Track of the MRIN-server starting at time t.

Hence,

$$\begin{aligned}
\frac{\text{MRIN}(\sigma)}{\text{OPT}(\sigma)} &= \frac{t + d(s^*(t),F) + d(F,x) + x}{\text{OPT}(\sigma)} + \frac{d(s(t),s^*(t))}{\text{OPT}(\sigma)}\\
&\leqslant 1 + \frac{d(s(t),s^*(t))}{\text{OPT}(\sigma)} && \text{by (6.17)}\\
&\leqslant \frac{4}{3} && \text{by (6.18).}
\end{aligned}$$

This proves the claim. □

Can one improve the competitiveness of $4/3$ or is this already best possible? Let us think for a moment: The problem with Algorithm MRIN is that shortly after it starts to return towards the origin from the furthest previously unserved request, a new request to the right of its server is released. In this case the MRIN-server has to return to a position it just left. Algorithm WS presented below avoids this pitfall successfully.

Algorithm 6.6 Algorithm WS ("Wait Smartly")

The WS-server moves right if there are yet unserved requests to the right of its present position. Otherwise, it takes the following actions. Suppose it arrives at its present position $s(t)$ at time t.

(i) Compute the optimal offline solution value $\text{OPT}(\sigma_{\leqslant t})$ for all requests released up to time t.

(ii) Determine a waiting time $W := \alpha\,\text{OPT}(\sigma_{\leqslant t}) - s(t) - t$, with $\alpha = (1+\sqrt{17})/4$.

(iii) Wait at point $s(t)$ until time $t+W$ and then start to move back to the origin.

One can prove the following result about the competitiveness of the more clever strategy WS:

Theorem 6.28 WS *is α-competitive for the* OLTSP *on $\mathbb{R}_{\geqslant 0}$ against a fair adversary with $\alpha = (1+\sqrt{17})/4 < 1.2808$.*

Proof: See [19, 5]. □

6.4.2 The Non-Abusive Adversary

An objective function that is very interesting in view of applications is the *maximum flow time*, which is defined to be the maximum time span from the release time of a request until the time it is served. We denote the OLTSP with this objective function as the F_{max}-OLTSP. Unfortunately, there can be no competitive algorithm, not even deterministic for the F_{max}-OLTSP as very simple constructions show.

The following lower bound result shows that even the fairness restriction on the adversary introduced in Section 6.4.1 is still not strong enough to allow for competitive algorithms in the F_{max}-OLTSP.

Theorem 6.29 *No randomized algorithm for the F_{max}-*OLTSP *on $\mathbb{R}$ can achieve a constant competitive ratio against an oblivious adversary. This result still holds, even if the adversary is* fair*, i.e., if at any moment in time t the server operated by the adversary is within the convex hull of the origin and the requested points from $\sigma_{\leqslant t}$.*

Proof: Let $\varepsilon > 0$ and $k \in \mathbb{N}$. We give two request sequences $\sigma_1 = (\varepsilon,\varepsilon),(2\varepsilon,2\varepsilon),\ldots,(k\varepsilon,k\varepsilon),(T,0)$ and $\sigma_2 = (\varepsilon,\varepsilon),(2\varepsilon,2\varepsilon),\ldots,(k\varepsilon,k\varepsilon),(T,k\varepsilon)$, each with probability $1/2$, where $T = 4k\varepsilon$.

The expected cost of an optimal fair offline solution is at most ε. Any deterministic online algorithm has cost at least $k\varepsilon/2$. The Theorem follows by applying Yao's Principle. □ □

The fair adversary is still too powerful in the sense that it can move to points where it knows that a request will appear without revealing any information to the online server before reaching the point. A *non-abusive* adversary does not possess this power.

Definition 6.30 (Non-Abusive Adversary)
An adversary ADV *for the* OLTSP *on $\mathbb{R}$ is* non-abusive*, if the following holds: At any moment in time t, where the adversary moves its server from its current position $p^{ADV}(t)$ to the right (left), there is a request from $\sigma_{\leqslant t}$ to the right (left) of $p^{ADV}(t)$ which* ADV *has not served yet.*

The following result shows that finally, the non-abusive adversary allows a distinction between different algorithms. The proof of the theorem is beyond the scope of these lecture notes and can be found in the cited paper.

Theorem 6.31 ([20]) *There exists a determinstic algorithm for the F_{max}-*OLTSP *which is 8-competitive against a non-abusive adversary.*

6.5 Exercises

Exercise 6.1
Show that the »natural REPLAN-approach« is not competitive for the $\sum w_j C_j$-OLDARP(this approach always recomputes an optimal solution with respect to the objective function of minimize the sum of completion times, if a new request becomes known).

Exercise 6.2
Prove Lemma 6.15: Let $a_i, b_i \in \mathbb{R}_{\geqslant 0}$ for $i = 1, \ldots, p$, for which

(i) $\sum_{i=1}^{p} a_i = \sum_{i=1}^{p} b_i$;

(ii) $\sum_{i=1}^{p'} a_i \geqslant \sum_{i=1}^{p'} b_i$ for all $1 \leqslant p' \leqslant p$.

Then $\sum_{i=1}^{p} \tau_i a_i \leqslant \sum_{i=1}^{p} \tau_i b_i$ for any nondecreasing sequence $0 \leqslant \tau_1 \leqslant \cdots \leqslant \tau_p$.

Exercise 6.3
Prove that MRIN is $3/2$-competitive for the OLTSP in $\mathbb{R}_{\geqslant 0}$.

7 Beyond Competitive Analysis

Even though competitive analysis has become the standard yardstick for online algorithms, it has also been critized a lot as being too crude and unrealistic. To prove that an online algorithm is c-competitive, we need to show that for *all* instances the algorithm does not perform worse than c times the optimum. That is, competitive analysis is a form of *worst case analysis*. The definition of a c-competitive algorithm and of the competitive ratio is both the weakness and strength of competitive analysis. It is a strength, because the setting is clear, the problems are crisp and the results are often elegant. But it is a weakness for several reasons.

First, in the face of of the devastatin comparison against an all-powerful cruel adversary (the optimal offline algorithm) a wide range of online algorithms (good, bad, or mediocre) can have the same competitive ratio; thus the competitive ratio is not very informative, as it fails to discriminate and to suggest good approaches. A good illustration for this is the paging problem (see Chapter 3). All marking algorithms, including both the good in practice LRU and the empirically mediocre FIFO, have the same competitive ratio.

Another aspect of the worst case analysis is that, since a worst case input decides the performance of an algorithm, the optimal algorithms are often unnatural and impractical. A good example of this is the result by Krumke et al. who consider the F_{max}-OLTSP [20]. In order to be competitive, they constructed an algorithm that might move into a direction where no request is.

Even enhancing the capabilities of an online algorithm, such as limited lookahead, does not help. Again the paging problem is a good example for this: A request sequence designed by a cruel adversary for an online algorithm without lookahead, can be transformed into a sequence for an algorithm that knows the next l request by replicating each request l times.

In this chapter, we discuss some approaches to overcome these difficulties. Koutsoupias and Papadimitriou proposed two methods [17]: the diffuse adversary and comparative analysis. Stochastic scheduling is another approach to deal with uncertainty in the area of scheduling problems. Finally, we discuss average case and the recently introduced smoothed competitive analysis [3].

7.1 Comparative Analysis

Instead of comparing an online algorithm to the optimal offline algorithm, in *comparative analysis* an algorithm is compared to a class of algorithms. If this class consists of all offline algorithms, obviously, comparative analysis coincides with competitive analysis.

Koutsoupias and Papadimitriou [17] considered a class of algorithms that only know the next l request in a metrical task system (see Chapter 4) or in the paging problem. For metrical task systems, they showed that an algorithm that does not know the next request, i.e., an online algorithm, has a comparative ratio to the class of algorithms with a lookahead of l, is at most $2l+1$. In the paging problem, the comparative ratio for online algorithms is $\min k, l+1$.

7.2 Diffuse Adversary

The diffuse adversary model, introduced in [17], restricts the power of the adversary. In the OLDARP (see Chapter 6), we have already seen some restrictions on the adversary. The adversary is only allowed to move within a certain part of the metric space. In the diffuse adverary model, the power of the adversary is restricted in a different way. We assume that the actual distribution, D, of the inputs is a member of a known class of possible distributions, Δ. The performance measure is now defined as

$$\max_{D \in \Delta} \frac{\mathbb{E}[\text{ALG}(D)]}{\mathbb{E}[\text{OPT}(D)]}$$

If the class of possible distribution consists of all distributions, then again, this model coincides with the standard competitive analysis: The diffuse adversary chooses (with probability 1) the input sequence generated by the cruel adversary.

Notice that this restriction on the adversary diverges from the restrictions on the adversaries in Chapter 6. For the latter, the adversary was charged with costs that can be higher than the optimal cost, whereas the diffuse adversary is charged with the (expected) optimal cost.

7.3 Stochastic Scheduling

Online optimization can be seen as a way to deal with uncertainty in optimization: An online algorithm has no knowledge at all of what will happen in the future. *Stochastic scheduling* is another way to deal with this uncertainty in the area of scheduling.

In the stochastic scheduling model, it is assumed that the processing time p_j of a job j is known in advance. It becomes known only upon completion of the job. However, the distribution of the corresponding random variable P_j is given beforehand. Another difference with online optimization is that also the number of jobs and the release time of these jobs are known. The solution of a stochastic scheduling problem is no longer a simple schedule, but a *scheduling policy*. Roughly spoken, a scheduling policy makes scheduling decisions at certain decision times t, and these decisions are based upon the observed past up to time t as well as the a priori knowledge of the input data of the problem. The policy, however, must not anticipate information about the future, such as the actual realizations p_j of the processing times of the jobs which have not yet been completed by time t.

Instead of comparing the expected performance of a scheduling policy to the expected value of the optimal solutions, in stochastic scheduling the performance measure is the ratio between the expected value of the scheduling policy to the expected value of the optimal *policy*. That is, stochastic scheduling addresses the ex-ante question "*What is the best that can be achieved under the given uncertainty about the future?*", whereas in competitive analysis the question is "*What was achieved under uncertainty about the future, and what could have been achieved if the future would not have been uncertain?*". For a more elaborate discussion on stochastic scheduling, we refer to [22].

Although, to the best of our knowledge, this stochastic approach has only been applied to scheduling problems, one can easily extend this approach to other problems in which uncertainty about the future occurs.

7.4 Average Case Competitive Analysis

Instead of considering the worst case instance for the competitive ratio of an algorithm, one can also try to make an average case analysis. For average case analysis, one needs to assume a probability distribution on the input instance, e.g. in the paging problem one may assume that a certain page i is asked with probability p_i. Furthermore, it is assumed that an online algorithm does not know this probability distribution.

One can then compare the expected value of the online algorithm to the expected value of the optimum. In contrast to stochastic scheduling, in average case analysis the optimal solution may be obtained by different algorithms for each possible outcome of the random process. Hence, the performance measure is

$$\frac{\mathbb{E}\left[\mathrm{ALG}(\sigma)\right]}{\mathbb{E}\left[\mathrm{OPT}(\sigma)\right]}. \tag{7.1}$$

An alternative measure is to consider the *expected competitive ratio*, i.e., the ratio between the algorithm and the optimum on every possible instance as a random variable (occuring with the probability that this instance occurs):

$$\mathbb{E}\left[\frac{\mathrm{ALG}(\sigma)}{\mathrm{OPT}(\sigma)}\right]. \tag{7.2}$$

Whereas the first performance measure tries to make a claim on the average performance of the algorithm compared to the average performance of the optimum, the expected competitive ratio says that the average ratio between the solution obtained by the algorithm to the optimal value is good or bad. The advantage of the expected competitive ratio is first that by applying Markov's Inequality, one can bound the probability that on an input instance the ratio is far away from the expectation. Hence, with high probability the expected competitive ratio shows the right order of magnitude of the competitive ratio. Second, the value of $\mathbb{E}\left[\mathrm{ALG}(\sigma)\right]$ may be dominated by a fraction of instances where the objective function is large. The behavior of the algorithm on the other instances is not or hardly taken into account. Using the measure in (7.2) all instances are weighted with $\mathrm{OPT}(\sigma)$ and have, thus, the same impact on the performance measure.

Using the expected competitive ratio, Scharbrodt et al. analyzed the average performance of a single machine scheduling problem so as to minimize the total completion time [27] and Fujiwara and Iwama analyzed the expected competitive ratio of a family of ski rental problems [12].

7.5 Smoothed Competitive Analysis

A disadvantage of of average case analysis is that it assumes a probability distribution over all input instances. To be able to do the analysis, this probability distribution normally has little resemblance with reality. To overcome the problems of average case and worst case analysis, Spielman and Teng introduced *smoothed complexity* [34]. Smoothed analysis is a hybrid between average case and worst case analysis, and was introduced to explain the success of algorithms that are known to work well in practice while presenting poor worst case performance.

The basic idea is to randomly perturb the initial input instances and to anlyze the average performance of the perturbed instances. The intuition is that the smoothed complexity is much smaller than its worst case complexity, if the worst case instances are isolated in the instance space. Becchetti et al. [3] extended the idea of Spielman and Teng to *smoothed competitive analysis*. The smoothed competitive analysis of an online algorithm is defined as

$$\sup \mathbb{E}_{\sigma \in N(\bar{\sigma})} \left[\frac{\mathrm{ALG}(\sigma)}{\mathrm{OPT}(\sigma)} \right],$$

where the supremum is taken over all input instances $\bar{\sigma}$, and the expectation is taken over all instances that are obtainable by smoothing the instance $\bar{\sigma}$ according to the smoothing model. As with "normal" competitive analysis, we can view smoothed competitive analysis as a game played by an online player against an adversary. However, in the smoothed case, the adversary is handicapped in the sense that after specifying the instance, some random mechanism changes his input. For smoothed analysis, one can define *oblivious* and *adaptive* adversaries, much in the same way as with randomized algorithms: an oblivious adversary needs to specify the complete input instance beforehand, whereas an (online) adaptive adversary is allowed to observe what has happened up to time t before deciding what request to release after time t. Notice that in this definition, we analyze the expected competitive ratio and not the ratio of expectations.

A main issue in smoothed analysis is to decide on what smoothing model to use. Several models have been proposed in the literature. Assume that (part of) the adversarial input instance is given as $\bar{I} = (\bar{x}_1, \ldots, \bar{x}_n)$, where the $\bar{x}_i$ is the part of the instance that is perturbed. We refer to $\bar{I}$ as the initial instance, and denote by $I = (x_1, \ldots, x_n)$ the smoothed or perturbed instance.

additive symmetric smoothing model. In the additive symmetric smoothening model each $\bar{x}_j$ is perturbed symmetrically around its initial value by adding some additive noise. That is, for each j, we choose an $\varepsilon_j \in [-L, L]$ independently at random and define

$$x_j = \bar{x}_j + \varepsilon_j,$$

where ε_j is drawn from a symmetric distribution with mean 0: $\Pr \varepsilon_j < -a = \Pr \varepsilon_j > a$ and $\mathbb{E}\left[\varepsilon_j\right] = 0$.

additive relative smoothing model. This model is similar to the previous one. The difference is that in the previous model, ε_j is choosen from the same interval, i.e., $[-L, L]$ for each j. In this model, we choose ε_j at random from an interval $[-L(\bar{x}_j), L(\bar{x}_j)]$, that is, the magnitude of ε_j depends on the initial value of input parameter j.

$$x_j = \bar{x}_j + \varepsilon_j,$$

where ε_j is drawn from a symmetric distribution with mean 0.

relative smoothing model. In the relative smoothing model, the input is smoothed symmetrically around its initial value by multiplying it with a random value:

$$x_j = \bar{x}_j (1 + \varepsilon_j),$$

where ε_j is drawn from a symmetric distribution with mean 0.

partial bit randomization model. This model is defined for integral values $\bar{x}_j$. This value is seen as a K-bit string and the k least significant bits are replaced by a random value:

$$x_j = 2^k \lfloor \frac{\bar{x}_j}{2^k} \rfloor + \varepsilon_j,$$

where $\varepsilon_j \in [0, 2^k - 1]$ is drawn randomly. Note that this smoothing model, is not, like the other models, a symmetric one, i.e., the expectation of x_j is not necessarily equal to $\bar{x}_j$.

Becchetti et al. considered a single machine scheduling problem so as to minimize the total flow time [3]. They showed that if the processing times are smoothed according to *partial bit randomization model*, the smoothed competitive ratio of the so-called Multi-Level Feedback algorithm exponentially decreases from the worst case lower bound to a constant. It can also be shown that smoothing release dates does not help: that is, when smoothing the release dates, the adversary is left with enough power to design an instance where the competitive ratio is the same as in the worst case.

Schäfer and Sivadasan [26] applied smoothed competitive analysis to WFA for metrical task systems (see Chapter 4).

A

Basic Probability Theory

In this appendix we review basic ideas from probability theory. We provide the basic definitions of *probability space*, *events*, *probability*, *independence*, *random variables*, and their *distributions* and *moments*. After these definitions, we prove some basic theorems, or state them without proof. We also address some tail inequalities, to bound probabilities that random variables deviate much from their expectations.

A.1 Basic Definitions

Any probability statement must refer to an underlying probability space. In the context of these lecture notes, this probability space is mostly clear without a formel specification. Moreover, we mostly consider discrete probability spaces, in contrast to continuous.

Definition A.1 (Probability space) *A (discrete)* ***probability space*** *is a pair* $(\Omega, \Pr)$*, where* Ω *denotes a non-empty (countable) set and* $\Pr : \Omega \to \mathbb{R}_+$ *is a* ***probability measure****, i.e., a function satisfying*

$$\sum_{\omega \in \Omega} \Pr[\omega] = 1.$$

The set Ω is also referred to as **sample space** and its elements are known as **elementary events**. A subset $\mathcal{E} \subseteq \Omega$ is called an **event**. The probability measure $\Pr$ can be extended to the set of all possible events, 2^{Ω}, by defining

$$\Pr[\mathcal{E}] = \sum_{\omega \in \mathcal{E}} \Pr[\omega].$$

Property A.2 *From Definition A.1 follows that a probability measure satisfies:*

(i) For every event $\mathcal{E} \subseteq \Omega$*, we have* $0 \leqslant \Pr[\mathcal{E}] \leqslant 1$*.*

(ii) For a countable set $\{\mathcal{E}_i : i \in I\}$ *of pairwise disjoint events, we have:* $\Pr[\bigcup_{i \in I} \mathcal{E}_i] = \sum_{i \in I} \Pr[\mathcal{E}_i]$*.*

Definition A.3 (Condition probability) *The* ***conditional probability*** *of* $\mathcal{E}_1$ *given* $\mathcal{E}_2$ *with* $\Pr[\mathcal{E}_2] > 0$ *is denoted by* $\Pr[\mathcal{E}_1 | \mathcal{E}_2]$ *and is given by*

$$\Pr[\mathcal{E}_1 | \mathcal{E}_2] = \frac{\Pr[\mathcal{E}_1 \cap \mathcal{E}_2]}{\Pr[\mathcal{E}_2]}$$

This corresponds to the probability that the outcome of a random experiment lies in the set $\mathcal{E}_1$, when we already know that it lies in the set $\mathcal{E}_2$.

Proposition A.4 *Let $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ be a partition of the sample space Ω. Then for any event $\mathcal{E}$,*

$$\Pr[\mathcal{E}] = \sum_{i=1}^{n} \Pr[\mathcal{E}|\mathcal{E}_1]\Pr[\mathcal{E}_i].$$

From Definition A.3 follows that $\Pr[\mathcal{E}_1 \cap \mathcal{E}_2] = \Pr[\mathcal{E}_1|\mathcal{E}_2]\Pr[\mathcal{E}_2] = \Pr[\mathcal{E}_2|\mathcal{E}_1]\Pr[\mathcal{E}_1]$, provided that $\Pr[\mathcal{E}_1] > 0$ as well as $\Pr[\mathcal{E}_2] > 0$.

Proposition A.5 (Bayes' rule) *Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be a partition of the sample space Ω. Then, for any event $\mathcal{E}$*

$$\Pr[\mathcal{E}_i|\mathcal{E}] = \frac{\Pr[\mathcal{E}_i \cap \mathcal{E}]}{\Pr[\mathcal{E}]} = \frac{\Pr[\mathcal{E}|\mathcal{E}_i]\Pr[\mathcal{E}_i]}{\sum_{j=1}^{n} \Pr[\mathcal{E}|\mathcal{E}_j]\Pr[\mathcal{E}_j]}.$$

Definition A.6 *Independence A collection of events $\{\mathcal{E}_i : i \in I\}$ is **independent** if for all subsets $S \subseteq I$*

$$\Pr[\bigcap_{i \in S} \mathcal{E}_i] = \prod_{i \in S} \Pr[\mathcal{E}_i].$$

*These events are said to be k-**wise independent** if every subcollection consisting of k events is independent. The special case of 2-wise independence is often referred to as **pairwise independence**.*

Equivalently, using the definition of conditional expectations, we can say that a collection of events $\{\mathcal{E}_i : i \in I\}$ is *independent* if for any $j \in I$ and all subsets $S \subseteq I$,

$$\Pr[\mathcal{E}_j|\cap_{i \in S}] = \Pr[\mathcal{E}_j].$$

In particular, if the events are *pairwise independent*, then $\Pr[\mathcal{E}_i|\mathcal{E}_j] = \Pr[\mathcal{E}_i]$, for all $i \neq j$.

Usually the events we deal with can be expressed in terms of real-valued functions called *random variables*.

Definition A.7 (Random variable) *A **random variable** X is a function $X : \Omega \to \mathbb{R}$. For $x \in \mathbb{R}$, $\Pr[X = x]$ denotes the probability of the event $\{\omega \in \Omega : X(\omega) = x\}$. Equivalently, for $x \in \mathbb{R}$, $\Pr[X \leqslant x]$ denotes the probability of the event $\{\omega \in \Omega : X(\omega) \leqslant x\}$.*

Definition A.8 (Distribution and density function) *The **distribution function** $F : \mathbb{R} \to [0,1]$ for a random variable X is defined as $F_X(x) = \Pr[X \leqslant x]$.*

*The **density function** $f : \mathbb{R} \to [0,1]$ for a random variable X is defined as $f_X(x) = \Pr[X = x]$.*

A *discrete random variable* is a function over the sample space whose range is either a finite or countably infinite subset of $\mathbb{R}$.

Definition A.9 (Expectation) *The **expectation** of a random variable X with density function f_X is defined as*

$$\mathbb{E}[X] = \sum_{x \in X(\Omega)} x f_X(x) = \sum_{x \in X(\Omega)} x \cdot \Pr[X = x].$$

For continuous random variables, the sum goes in the limit into an integral

$$\mathbb{E}[X] = \int_{x \in X(\Omega)} x f_X(x) \mathrm{d}x.$$

For any two random variables X and Y, we have that $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$, even if these random variables are not independent. In fact, we can generalize this as follows.

Proposition A.10 (Linearity of expectation) *Let* $X_1, \ldots, X_n$ *be arbitrary random variables, and let* $h(X_1, \ldots, X_n)$ *be a linear function. Then*

$$\mathbb{E}[h(X_1, \ldots, X_n)] = h(\mathbb{E}[X_1], \ldots, \mathbb{E}[X_n]).$$

This result does not generalize to nonlinear functions, although with the assumption of independence, we can prove a similar result for any polynomial h using the following.

Proposition A.11 *For* independent *random variables* X *and* Y*,*

$$\mathbb{E}[XY] = \mathbb{E}[X]\,\mathbb{E}[Y].$$

Some other useful properties of expectations are given in the following proposition. We say that a random variable X *stochastically dominates* a random variable Y if, for all $z \in \mathbb{R}$, $\Pr[X > z] \geqslant \Pr[Y > z]$.

Proposition A.12 *Let* X *and* Y *be random variables with finite expectation.*

(i) If X *stochastically domninates* Y*, then* $\mathbb{E}[X] \geqslant \mathbb{E}[Y]$*; equality holds if* X, Y *are identically distributed.*

(ii) $|\mathbb{E}[X]| \leqslant \mathbb{E}[|X|]$.

(iii) For a non-negative integer-valued random variable X, $\mathbb{E}[X] = \sum_{x=0}^{\infty} \Pr[X \geqslant x]$.

The expectation of a random variable is also referred to as the *first moment* of this random variable.

Definition A.13 (Moments) *For* $k \in \mathbb{N}$*, the* k***th moment,*** m_X^k *and the* k***th central moment*** μ_X^k *of a random variable* X *are defined as*

$$\begin{aligned} m_X^k &= \mathbb{E}\left[X^k\right], \\ \mu_X^k &= \mathbb{E}\left[(X - \mathbb{E}[X])^k\right]. \end{aligned}$$

To confuse the notation, μ_X is often used for the expected value of X, i.e., $\mu_x = m_X^1$, whereas $\mu_X^1 = 0$. The second central moment is known as *the variance* of X and is denoted by $\mathrm{Var}[X]$ of σ_X^2. *The standard deviation*, denoted by σ_X, is the positive square root of the variance.

Proposition A.14

$$Var[X] = m_X^2 - (\mu_X)^2 = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2.$$

Proposition A.15 *For independent random variables* X *and* Y*,*

$$Var[X+Y] = Var[X] + Var[Y].$$

This proposition cannot be generalized to arbitrary functions, since $\mathrm{Var}[cX] = c^2 \mathrm{Var}[X]$.

A.2 Standard Probability Distributions

Now, we have introduced some basic terminology and propositions, we described some common probability distributions.

Uniform distribution. The uniform distribution is a commonly used distribution for the probabilistic analysis of combinatorial optimization algorithms and online algorithms. In the *discrete* uniform distribution, we have a finite set Ω, e.g., the set of all possible instances, and the probability is for each element ω equal, i.e., $\Pr[\omega] = \frac{1}{|\Omega|}$, for $\omega \in \Omega$.

In the *continuous* uniform distribution, we assume that we pick at random a point in an interval $[a,b]$. The probability density function for each point is $f_X(x) = \frac{1}{b-a}$. The expected value of a random variable X uniformly distributed on the interval $[a,b]$ is $\mathbb{E}[X] = \frac{a+b}{2}$. The variance for this random variable is $\mathrm{Var}[X] = \frac{(b-1)^2}{12}$.

Bernoulli distribution. Suppose we flip a coin whose probability of HEADS is p. Let X be a random variable that has value 1 if the result is HEADS, and 0 otherwise. Then X has the Bernoulli distribution with parameter p. The density function for X is given by

$$f_X(x) = \begin{cases} 1-p & \text{if } x = 0, \\ p & \text{if } x = 1, \\ 0 & \text{otherwise.} \end{cases}$$

The expectation of X is $\mathbb{E}[X] = p$, and the variance is $\mathrm{Var}[X] = p(1-p)$.

Binomial distribution. Let $X_1, \ldots, X_n$ be independent identically distributed random variables whose common distribution is the Bernoulli distribution with parameter p. The random variable $X = X_1 + \ldots + X_n$ denotes the number of HEADS in a sequence of n coin flips. The random variable X has the binomial distribution with parameter n and p, sometimes abbreviated $B(n,p)$. The density function is denoted by $b(k;n,p)$, and for integer k with $0 \leqslant k \leqslant n$, we have

$$b(k;n,p) = \Pr X = k = \binom{n}{k} p^k (1-p)^{n-k}.$$

The expectation of X is $\mathbb{E}[X] = np$ and the variance is $\mathrm{Var}[X] = np(1-p)$.

Geometric distribution. Suppose we flip a coin repeatedly until HEADS appears for the first time. Assuming that each coin flip has the Bernoulli distribution with parameter p, the random variable X denoting the total number of coin flips has the geometric distribution with parameter p. Its density function is as follows.

$$\Pr X = k = p(1-p)^{k-1}.$$

The expectation of a geometric X is $\mathbb{E}[X] = 1/p$ and the variance is $\mathrm{Var}[X] = q/p^2$

Exponential distribution. Let X be a continuous random variable from the exponential distribution with parameter $\lambda > 0$. The probability distribution function of X is

$$f(x,\lambda) = \lambda e^{\lambda x}, \quad x > 0.$$

The cumulative distribution function is

$$\Pr X \leqslant x = 1 - e^{-\lambda x}, \quad x > 0.$$

The expected value is $\mathbb{E}[X] = 1/\lambda$ and the variance is $\mathrm{Var}[X] = 1/\lambda^2$.

A.3 Tail probabilities

In many applications we need to bound the probability that a random variable is far away from its expectation. In this section, we give some useful inequalities for this.

Theorem A.16 (Markov Inequality) *Let X be a random variable asuuming only non-negative values. Then for all $t \in \mathbb{R}_+$,*

$$\Pr X \geqslant t\mathbb{E}[X] \leqslant \frac{1}{t}.$$

Proof: Define a function $f(x)$ by $f(x) = 1$ if $x \geqslant t\mathbb{E}[X]$, and 0 otherwise. Then $\Pr X \geqslant t\mathbb{E}[X] = \mathbb{E}[f(X)]$. Since, $f(x) \leqslant (x/t\mathbb{E}[X])$ for all x,

$$\mathbb{E}[f(X)] \leqslant \mathbb{E}\left[\frac{X}{t\mathbb{E}[X]}\right] = \frac{\mathbb{E}[X]}{t\mathbb{E}[X]} = \frac{1}{t}$$

and the theorem follows. □

Theorem A.17 (Chebyshev's Inequality) *Let X be a random variable with expectation μ_X and standard deviation σ_X. Then for any $t \in \mathbb{R}_+$,*

$$\Pr|X - \mu_X| \geqslant t\sigma_X \leqslant \frac{1}{t^2}.$$

Proof: First, note that

$$\Pr|X - \mu_X| \geqslant t\sigma_X = \Pr(X - \mu_X)^2 \geqslant t^2\sigma_X^2.$$

The random variable $Y = (X - \mu_X)$ has expectation σ_X^2 and only assumes non-negative values. Hence, we can apply the Markov inequality to bound this probability by $\frac{1}{t^2}$. □

Theorem A.18 (Chernoff bounds) *Let $X_1, \ldots, X_n$ be independent Bernoulli random variables such that for $1 \leqslant i \leqslant n$, $\Pr X_i = 1 = p_i$, where $0 < p_i < 1$. Then for $X = X_1 + \ldots + X_n$, $\mu = \mathbb{E}[X] = p_1 + \ldots p_n$, and any $\delta > 0$,*

$$\begin{aligned} \Pr X > (1+\delta)\mu &< \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu, \\ \Pr X < (1-\delta)\mu &< \exp(-\mu\delta^2/2) \end{aligned}$$

Bibliography

[1] E. J. Anderson and C. N. Potts, *On-line scheduling of a single machine to minimize total weighted completion time*, Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 548–557.

[2] G. Ausiello, E. Feuerstein, S. Leonardi, L. Stougie, and M. Talamo, *Algorithms for the on-line traveling salesman*, Algorithmica **29** (2001), no. 4, 560–581.

[3] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, G. Schäfer, and T. Vredeveld, *Average case and smoothed competitive analysis of the multi-level feedback algorithm*, Proceedings of the 44th Annual IEEE Symposium on the Foundations of Computer Science, 2003, pp. 462–475.

[4] S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson, *On the power of randomization in on-line algorithms*, Algorithmica **11** (1994), 2–14.

[5] M. Blom, S. O. Krumke, W. E. de Paepe, and L. Stougie, *The online-TSP against fair adversaries*, Informs Journal on Computing **13** (2001), no. 2, 138–148, A preliminary version appeared in the Proceedings of the 4th Italian Conference on Algorithms and Complexity, 2000, vol. 1767 of Lecture Notes in Computer Science.

[6] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*, Cambridge University Press, 1998.

[7] E. Davis and J. M. Jaffe, *Algorithms for scheduling tasks on unrelated processors*, Journal of the ACM **28** (1981), no. 4, 721–736.

[8] D. R. Dooley, S. A. Goldman, and S. D. Scott, *TCP dynamic acknowledgement delay: Theory and practice*, Proceedings of the 30th Annual ACM Symposium on the Theory of Computing, 1998, pp. 389–398.

[9] L. Epstein and R. van Stee, *Lower bounds for on-line single-machine scheduling*, Theoretical Computer Science **299** (2003), no. 1–3, 439–450.

[10] A. Fiat, Y. Rabani, and Y. Ravid, *Competitive* k*-server algorithms*, Proceedings of the 31st Annual IEEE Symposium on the Foundations of Computer Science, 1990, pp. 454–463.

[11] A. Fiat and G. J. Woeginger (eds.), *Online algorithms: The state of the art*, Lecture Notes in Computer Science, vol. 1442, Springer, 1998.

[12] H. Fujiwara and K. Iwama, *Average-case competititve analyses for ski-rental problems*, Algorithmica **42** (2005), no. 1, 95–107.

[13] R. L. Graham, *Bounds for certain multiprocessing anomalies*, Bell System Technical Journal **45** (1966), 1563–1581.

[14] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, *Optimization and approximation in deterministic sequencing and scheduling: a survey*, Annals of Discrete Mathematics **5** (1979), 287–326.

[15] Ronald L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics **17** (1969), 263–269.

[16] H. Hoogeveen and A. P. A. Vestjens, *Optimal online algorithms for single-machine scheduling*, Proceedings of the 5th Mathematical Programming Society Conference on Integer Programming and Combinatorial Optimization, Lecture Notes in Computer Science, vol. 1084, 1996, pp. 404–414.

[17] E. Koutsoupias and C. Papadimitriou, *Beyond competitive analysis*, Proceedings of the 35th Annual IEEE Symposium on the Foundations of Computer Science, 1994, pp. 394–400.

[18] ———, *On the* k*-server conjecture*, Journal of the ACM **42** (1995), no. 5, 971–983.

[19] S. O. Krumke, *Online optimization: Competitive analysis and beyond*, Habilitationsschrift, Technische Universität Berlin, 2002.

[20] S. O. Krumke, L. Laura, M. Lipmann, A. Marchetti-Spaccamela, W. E. de Paepe, D. Poensgen, and L. Stougie, *Non-abusiveness helps: An* $\mathrm{O}(1)$*-competitive algorithm for minimizing the maximum flow time in the online traveling salesman problem*, Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization, Lecture Notes in Computer Science, vol. 2462, Springer, 2002, pp. 200–214 (english).

[21] X. Lu, R. Sitters, and L. Stougie, *A class of on-line scheduling algorithms to minimize total completion time*, Operations Research Letters **31** (2003), no. 3, 232–236.

[22] R. H. Möhring, F. J. Radermacher, and G. Weiss, *Stochastic scheduling problems I: General strategies*, ZOR - Zeitschrift für Operations Research **28** (1984), 193–260.

[23] R. Motwani and P. Raghavan, *Randomized algorithms*, Cambridge University Press, 1995.

[24] C. A. Phillips, C. Stein, and J. Wein, *Minimizing average completion time in the presence of release dates*, Mathematical Programming **82** (1998), 199–223.

[25] S. Sahni and Y. Cho, *Scheduling independent tasks with due times on a uniform processor system*, Journal of the ACM **27** (1980), no. 3, 550–563.

[26] G. Schäfer and N. Sivadasan, *Topology matters: Smoothed competitiveness of metrical task systems*, Proceedings of the 21th International Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 2996, Springer, 2004, pp. 489–500.

[27] M. Scharbrodt, Th. Schickinger, and A. Steger, *A new average case analysis for completion time scheduling*, Journal of the ACM **53** (2006), no. 1, 121–146.

[28] W. E. Schmith, *Various optimizers for single stage production*, Navel Resarch and Logistics Quarterly **3** (1956), 59–66.

[29] L Schrage, *A proof of the optimality of the shortest remaining processing time discipline*, Operations Research **16** (1968), 199–223.

[30] A. S. Schulz and M. Skutella, *The power of alpha-points in preemptive single machine scheduling*, 1999.

[31] J. Sgall, *Online scheduling*, in Fiat and Woeginger [11].

[32] R. Sitters, *Complexity and approximation in routing and scheduling*, PhD thesis, Eindhoven University of Technology, 2004.

[33] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM **28** (1985), no. 2, 202–208.

[34] D. A. Spielman and S.-H. Teng, *Smoothed analysis: Why the simplex algorithm usually takes polynomial time*, Journal of the ACM **51** (2004), no. 3, 386–463.

[35] A. P. A. Vestjens, *On-line machine scheduling*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.