Lecture Notes

# Online Optimization

Yann Disser
TU Darmstadt
winter 2018/19

# Contents

# 1 Introduction

In classical (*offline*) optimization problems we are given some input data and need to make optimization decisions (i.e., set variables) such that a certain objective is optimized. This framework is too restrictive for problems where parts of the input become available over time (i.e., *online*) and only after some optimization decisions already had to be fixed. For example, consider the problem of controlling a personal elevator: People arrive over time on different floors and request the elevator, but the elevator cannot wait for all requests to arrive before starting to move, at least if we want to achieve a good completion time until all requests have been served.

*Online optimization* addresses problems with a temporal component as in the elevator problem above. Conceptually, an online algorithm gets its input item by item, one input *event* at a time (e.g., elevator button press). Upon each event, the algorithm immediately needs to respond by making an optimization decision (e.g., move the elevator up/down or wait). Importantly, the algorithm needs to act on the same timeline ("online") on which events occur, i.e., each decision must only be based on past events and must be independent of the future. The challenge for the online algorithm lies in finding a good balance between efficiently handling events while protecting itself against possible futures. For example, assume the elevator is at the ground floor and receives a request in the fifth floor. Should it start moving immediately, or should it wait a bit to see whether there are passengers that want to go upwards from the ground floor?

In this lecture, we devise "good" online algorithms for various problems. Of course, no online algorithm can always be good, because no matter how it decides to react to a request, its response can always turn out to be a mistake in retrospect. A possible quality measure for an online algorithm would be to consider its expected objective function value with respect to a fixed distribution over inputs. The main issue with this approach lies in how to choose a suitable distribution, and that the fact whether an algorithm is good heavily depends on this choice. Instead, we will use *competitive analysis,* an absolute quality measure for online algorithms based on their worst-case behavior, much like classical running time analysis concentrates on worst-case running times.

Competitive analysis challenges an online algorithm to compete against an *adversary* that counters every action of the algorithm with a worst-possible sequence of future events. For example, if the elevator decides to move away from the ground floor immediately, the adversary may decide to send a request at the ground floor in the next time step, and if the elevator waits, the adversary may not send additional requests. As mentioned above, the adversary can often easily ruin the objective function value of the online algorithm. To make the interaction fairer for the online algorithm, we demand that the adversary present its own *offline* solution once all requests have been revealed, and we compare the performance of the online algorithm with the quality of the offline solution by computing the *competitive ratio* between the two. Of course, this measure is still quite harsh for the online algorithm, since the offline solution of the adversary may use full knowledge of the entire event sequence. However, in many cases, we will be able to give online algorithms with bounded competitive ratios. Lower bounds on the best-possible competitive ratio of an online problem express the loss in solution quality that is due to imperfect information in the online setting, and we will consider an online algorithm to be "good", if its competitive ratio is best-possible.

In the following, we formally define the notions introduced above and apply competitive analysis to various typical online problems. These online problems naturally arise in many settings:

**Server Problems:** Requests arrive over time and we control server movements (e.g., elevators).

**Data Reorganization:** Data is accessed over time and we have to reorganize a data structure to speed up future accesses.

**Machine Scheduling:** Tasks arrive over time and we have to assign them to machines (workers).

**Bin Packing:** Items arrive over time and we have to pack them into bins of limited size.

**Bipartite Matching:** Candidates arrive over time and we have to assign each to a partner.

**Trading:** Stock prices arrive over time and we have to decide whether to buy or sell stocks.

## 1.1 Offline optimization

We start by revisiting offline optimization and formally establish the notation we will use in this lecture.

**Definition 1.1.** An *(offline) optimization problem* $\mathcal{P}$ is a 4-tuple $(\mathcal{I}, \mathrm{Sol}, c, \mathrm{type})$, where:

- $\mathcal{I}$ is a set of input instances
- Sol is a function mapping each instance $I \in \mathcal{I}$ to a set of feasible solutions with $\mathcal{S} := \cup_{I \in \mathcal{I}} \mathrm{Sol}(I)$.
- $c \colon \mathcal{I} \times \mathcal{S} \to \mathbb{R}$ is the objective function (*cost* or *utility*)
- $\mathrm{type} \in \{\min, \max\}$ expresses whether we want to minimize the cost $c$ or maximize the utility $c$

Most optimization problems we consider are minimization problems, and, for clarity, the following definitions are with respect to a fixed optimization problem $\mathcal{P} = (\mathcal{I}, \mathrm{Sol}, c, \min)$, where $c$ is a cost function. We introduce basic notation to distinguish between the solutions computed by an algorithm Alg and its cost.

**Definition 1.2.** An *(offline) algorithm* Alg computes a feasible solution $\textsc{Alg}[I] \in \mathrm{Sol}(I)$ of cost $\textsc{Alg}(I) := c(I, \textsc{Alg}[I])$ for all $I \in \mathcal{I}$.

**Definition 1.3.** By Opt we denote an (arbitrary but fixed) *optimum* algorithm with $\textsc{Opt}(I) = \min_{S \in \mathrm{Sol}(I)} c(I, S)$ for all $I \in \mathcal{I}$.

The definition of a competitive ratio will be similar to the approximation ratio of approximation algorithms, which we recall below.

**Definition 1.4.** A polynomial-time algorithm Alg is an *(asymptotic) $\rho$-approximation algorithm* (with constant $\alpha \geq 0$) if for all $I \in \mathcal{I}$

$$\textsc{Alg}(I) \leq \rho \cdot \textsc{Opt}(I) \ (+\alpha).$$

**Definition 1.5.** The *approximation ratio* of a polynomial-time algorithm Alg is the infimum over all $\rho$, such that Alg is a $\rho$-approximation algorithm.

## 1.2 Online optimization

We now make the notion of online algorithms formal. Intuitively, instead of being provided with the entire input at once, online algorithms get a sequence of inputs (requests) and need to react to (i.e., to answer) each request immediately. Formally, we express this interaction as a game in the following sense.

**Definition 1.6.** An *online problem* is expressed as a *request-answer game* $(R, \Sigma, A, \mathcal{C})$, where:

- $R$ is the set of possible requests
- $\Sigma \subseteq \bigcup_{i \in \mathbb{N}} R^i$ is the set of all possible instances
- $A$ is the set of possible answers
- $\mathcal{C} = (c_i)_{i \in \mathbb{N}}$ is a sequence of cost (utility) functions $c_i \colon R^i \times A^i \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ that capture the cost (utility) of the answers up to time step $i$ and $c_0 := 0$

For an instance $\sigma = (r_1, \ldots, r_n) \in \Sigma$ of length $n \in \mathbb{N}$ we write $\sigma_{\leq i} := (r_1, \ldots, r_i)$ and $\sigma_{<i} := \sigma_{\leq i-1}$ for $i \leq n$ and $\sigma_{\leq 0} := \emptyset$. We call an online problem *finite* if $R, A, \Sigma$ are all finite.

Our definition of an online algorithm needs to allow for sequential input, but at the same time ensures that requests are answered immediately, i.e., without using knowledge of future requests. The following definitions are with respect to a fixed request-answer game $(R, \Sigma, A, \mathcal{C})$, and we again assume that $c_1, c_2, \dots$ are cost functions that need to be minimized, the definitions for utility maximization instead of cost minimization are analogous.

**Definition 1.7.** A *deterministic online algorithm* ALG computes a sequence $(f_i \colon R^i \to A)_{i\in\mathbb{N}}$. The *solution* of ALG for the sequence $\sigma = (r_1, \dots, r_n) \in (\Sigma \cap R^n)$ is $\text{ALG}[\sigma] := (a_1, \dots, a_n) \in A^n$, where $a_i := f_i(\sigma_{\leq i})$, its total cost is $\text{ALG}(\sigma) := c_n(\sigma, \text{ALG}[\sigma])$, and its cost in step $i \in \{1, \dots, n\}$ is $\text{ALG}_i(\sigma) := \text{ALG}(\sigma_{\leq i}) - \text{ALG}(\sigma_{\leq i-1})$.

As a benchmark, we will compare the performance of an online algorithm on an instance to the best possible offline solution we could have gotten if we had known the entire request sequence ahead of time.

**Definition 1.8.** The *(offline) optimum solution* $\text{OPT}[\sigma]$ for sequence $\sigma = (r_1, \dots, r_n) \in (\Sigma \cap R^n)$ is fixed such that $\text{OPT}[\sigma] \in \arg\min_{\mathbf{a}\in A^n} c_n(\sigma, \mathbf{a})$. Its total cost is $\text{OPT}(\sigma) = c_n(\sigma, \text{OPT}[\sigma])$. The optimum cost in step $i \in \{1, \dots, n\}$ is defined via $\text{OPT}_i(\sigma) := \text{OPT}(\sigma_{\leq i}) - \text{OPT}(\sigma_{\leq i-1})$.

Notice that, in contrast to the definition of online algorithms, OPT need not be consistent between time steps. In particular, the offline optimum can be completely different for instances $\sigma_{\leq n-1}$ and $\sigma_n$.

We are now ready to introduce the competitive ratio that we will use throughout this lecture to assess the quality of online algorithms. Observe that the definition is unfair in the sense that we do not compare to the best possible online algorithm, but to the best offline solution for each instance independently, even without requiring the solutions to these instances to be consistent with one another.

**Definition 1.9.** A deterministic online algorithm ALG is *$\rho$-competitive* if there is a constant $\alpha \geq 0$, such that for all instances $\sigma \in \Sigma$

$$\text{ALG}(\sigma) \leq \rho \cdot \text{OPT}(\sigma) + \alpha.$$

ALG is *strictly $\rho$-competitive* if the above holds for $\alpha = 0$.

Note that, in general, we allow $\rho$ to be any function of $\sigma$ (in particular of the length $n$ of the input) and of any other problem parameters (e.g., number of floors of the building for the elevator).

**Definition 1.10.** If there is a constant $\rho \in \mathbb{R}$ for which the online algorithm ALG is (strictly) $\rho$-competitive, we say that ALG is *(strictly) competitive*. The *(strict) competitive ratio* of ALG then is the infimum over all $\rho \in \mathbb{R}$, such that ALG is (strictly) $\rho$-competitive.

**Definition 1.11.** The *(strict) competitive ratio* of a problem is the infimum over all $\rho$ for which a (strictly) $\rho$-competitive algorithm exists.

Note similarities and differences to the notion of approximation ratios for the offline setting. In both notions, we ask how close we can get to optimal solutions in the worst-case if we have limited resources. For approximation algorithms, the allowed processing time is limited, while online algorithms have limited access to information. While we do not require online algorithms to be efficient, most algorithms we will see will have polynomial running times.

Also note that while the strict competitive ratio is a guarantee for every possible request sequence, the (non-strict) competitive ratio provides an asymptotic guarantee. In order to show lower bounds on the competitive ratio of an algorithm, we will often use the following characterization.

**Proposition 1.12.** *An online algorithm* ALG *has competitive ratio greater than $\rho$ if and only if there exists an infinite sequence $(\sigma^{(i)} \in \Sigma)_{i\in\mathbb{N}}$ with $\lim_{i\to\infty} \text{ALG}(\sigma^{(i)}) = \infty$ and*

$$\limsup_{i\to\infty} \frac{\text{ALG}(\sigma^{(i)})}{\text{OPT}(\sigma^{(i)})} > \rho.$$

## 1.3 The ski rental problem

As an example to see the above definitions in action, we consider a typical everyday problem: If we do not know how often (or long) we will use some item, for how long should we rent it before deciding to buy? This problem is typically phrased in terms of renting skis, but can trivially be adapted to many similar situations. While, formally, we would have to specify the individual components (request set, answer sets, cost functions) of the underlying request answer game, we will usually resort to a more intuitive description of online problems in the following standard form:

| Ski Rental Problem | |
|---|---|
| **given:** | cost 1 to rent skis for one day, cost $B \in \mathbb{N}$ to buy skis |
| **online:** | days $\sigma = (d_1, \dots, d_n) \in \{1\}^n$ that skis are needed (only $n$ is unknown) |
| **actions** (step $i$)**:** | rent skis (*cost* 1) **or** buy skis (*cost* $B$) **or** use skis bought earlier (*cost* 0) |
| **objective:** | minimize the total cost $\sum_{i=1}^{n} \textsc{Alg}_i(\sigma)$ |

For the sake of the example, we will, for once, also give the formal definition of the problem. Since only the rental duration is unknown, we have a trivial set of requests $R = \{\text{need skis}\}$ that only allows for the request "need skis" for every day that skis are needed. Then $\Sigma = \bigcup_{i\in\mathbb{N}} R^i$. The possible answers on each day are $A = \{\text{rent}, \text{buy}, \text{use}\}$. At this point it may be worrying that we allow "use" without ensuring that we previously chose "buy". Constraints that are based on the previous answers generally need to be modeled via the cost functions. We let $c_0 := 0$ and define

$$c_i(a_1, \dots, a_i) = c_{i-1}(a_1, \dots, a_{i-1}) + \begin{cases} 1, & \text{if } x = \text{rent}, \\ B, & \text{if } x = \text{buy}, \\ 0, & \text{if } x = \text{use} \wedge \text{buy} \in \cup_{j=1}^{i-1}\{a_j\}, \\ \infty, & \text{otherwise.} \end{cases}$$

Note that the only information that an online algorithm learns in step $i$ is that $n \geq i$. Therefore, every (sensible) deterministic online algorithm can be expressed as an algorithm $\textsc{Alg}^i$ that always buys in the $i$-th step or the algorithm $\textsc{Alg}^\infty$ that never buys. We show that the unique best strategy (in the worst-case) for renting skis is to buy in step $B$.

**Theorem 1.13.** *$\textsc{Alg}^B$ is strictly $(2 - 1/B)$-competitive.*

*Proof.* First observe that the optimum cost is given by

$$\textsc{Opt}(\sigma) = \min\{n, B\}.$$

The cost of $\textsc{Alg}^B$ is

$$\textsc{Alg}^B(\sigma) = \begin{cases} n & \text{if } n < B, \\ 2B - 1 & \text{else.} \end{cases}$$

Now, if $n < B$, then

$$\textsc{Alg}^B(\sigma) = n = \textsc{Opt}(\sigma),$$

and if $n \geq B$, then

$$\textsc{Alg}^B(\sigma) = 2B - 1 = (2 - 1/B) \cdot \textsc{Opt}(\sigma).$$

Overall, we get $\textsc{Alg}^B(\sigma) \leq (2 - 1/B) \cdot \textsc{Opt}(\sigma)$ as claimed. □

**Theorem 1.14.** *For any $i \neq B$, $\textsc{Alg}^i$ is not strictly $(2 - 1/B)$-competitive.*

*Proof.* We want to prove a lower bound on the competitive ratio of $\textsc{Alg}^i$ for $i \neq B$, so we take the perspective of the adversary that reacts with the worst-possible request sequence to the answers of the algorithm. The only freedom the adversary has in the ski rental problem lies in the choice of $n$. Since $\textsc{Alg}^\infty(\sigma) \to_{n\to\infty} \infty$ while $\textsc{Opt}(\sigma) \leq \min\{n, B\}$, trivially, $\textsc{Alg}^\infty$ is not competitive (cf. Proposition 1.12).

For finite $i$, we consider the most natural strategy (for the adversary) that stops the request sequence just after $\text{ALG}^i$ buys, i.e., just after step $i$.

If $i \leq B - 1$, we have

$$\text{ALG}^i(\sigma) = i - 1 + B \geq 2i = 2 \cdot \text{OPT}(\sigma).$$

If $i \geq B + 1$, we have

$$\text{ALG}^i(\sigma) = i - 1 + B \geq 2B = 2 \cdot \text{OPT}(\sigma).$$

In both cases $\text{ALG}^i(\sigma) \geq 2 \cdot \text{OPT}(\sigma) > (2 - 1/B) \cdot \text{OPT}(\sigma)$, as claimed. □

# 2 The Paging Problem

A typical source of online problems is the dynamic re-organization of data structures in order to improve access times for elements that are accessed often. The paging problem is a basic online problem of this type that models data access in memory hierarchies, where memory layers close to the CPU are very fast but small, while memory layers further away are much slower but also much larger. In the paging problem we consider a simple hierarchy consisting of a slow main memory that contains $m$ memory pages and a fast cache that can only hold $k < m$ pages at the same time (see Figure 1). The problem now is to decide online which pages to keep in the cache over time in order to minimize the number of accesses to main memory.

| PAGING PROBLEM | |
|---|---|
| **given:** | memory pages $P = \{p_1, \ldots, p_m\}$, cache size $k \in \mathbb{N}$,<br>cache $C$ with initial contents $C_0 \subset P$, $\lvert C_0 \rvert = k$ |
| **online:** | sequence $\sigma = (r_1, \ldots, r_n) \in P^n$ of pages that are accessed in this order |
| **actions** (step $i$)**:** | evict (remove) pages from $C$ (**free**)<br>load any number $m_i \leq k - \lvert C \rvert$ of pages from $P \setminus C$ into $C$ (**paid**)<br>at the end of step $i$, we must have $r_i \in C$ |
| **objective:** | minimize the total cost $\sum_{i=1}^{n} \text{ALG}_i(\sigma) = \sum_{i=1}^{n} m_i$ |

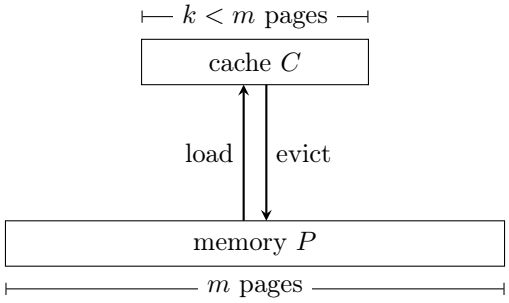


Figure 1: Illustration of the paging memory model with a fast/small cache $C$ and a slow/large memory $P$.

*Remark* 2.1. We can restrict ourselves to *demand paging* algorithms that may only load page $r_i$ in step $i$, and only evict a page if needed to make room for $r_i$. We can easily bring any algorithm into this form by delaying all unnecessary evicts and loads for as long as possible. Dealing with demand paging algorithms simplifies the analysis, because we only need to count the number of *page faults*, i.e., the number of times a page in $P \setminus C$ is requested.

## 2.1 Optimal offline paging

If we know the sequence of pages that will be requested ahead of time, it is natural to use the *longest-forward-distance* (LFD) heuristic for the paging problem: On every page fault evict any page that will not be requested again, and, if no such page exists, evict the page whose next access is furthest in the future. As an example, consider the request sequence $\sigma = (a, a, b, e, f, b, b, c, d, a, b)$ for the cache $C_0 = \{a, b, c, d\}$ of capacity $k = 4$. The first cache fault is caused by $r_4 = e$, and LFD evicts $a$ (forward distance 6) and replaces it by $e$.

**Theorem 2.2** (Belady [1966])**.** LFD *is an optimal (offline) paging algorithm.*

*Proof.* For the sake of contradiction, assume that LFD is not optimal and let $\sigma = (r_1, \ldots, r_n)$ be an instance where it is not optimal. Let OPT be an optimal (demand paging) algorithm that maximizes

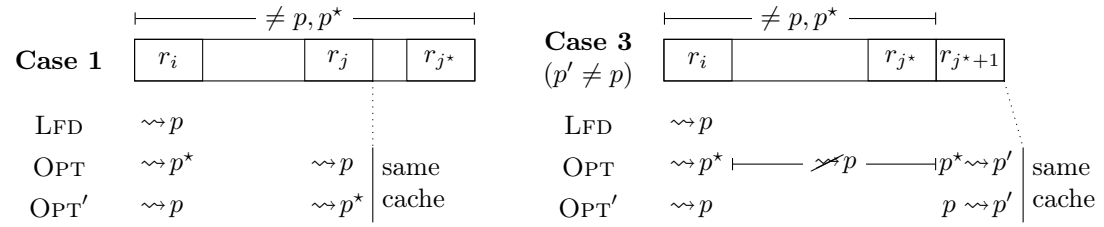


Figure 2: Illustration of cases 1 and 3 in the proof of Theorem 2.2. We use '$x \rightsquigarrow y$' to denote that page $x$ is loaded and page $y$ is evicted.

the number of steps $i$ until LFD and OPT make a different decision, i.e., OPT and LFD make the same decisions on $\sigma_{\leq i-1}$ but evict a different page for request $r_i$. We construct an optimum solution OPT$'[\sigma]$ that behaves like LFD on $\sigma_{\leq i}$, which then contradicts our choice of OPT and therefore completes the proof. Note that the solution OPT$'[\sigma]$ that we construct is only demand paging up to step $i$, but can easily be modified as described in Remark 2.1 to be demand paging for all steps, without changing its behavior in the first $i$ steps.

Let $p$ be the page evicted by LFD in step $i$ and $p^\star \neq p$ be the page evicted by OPT in the same step. We define $j^\star = \max\{j \in \{i, \ldots, n\} \colon p^\star \notin \{r_i, r_{i+1}, \ldots, r_j\}\}$ to be the last consecutive step after step $i$, where $p^\star$ is not requested. By definition of LFD, we know that $p$ is not requested in steps $\{i, \ldots, j^\star\}$. We distinguish three cases (see Figure 2):

**Case 1: $p$ is evicted by OPT in some step $j \in \{i+1, \ldots, j^\star\}$.**
Since OPT is demand paging and since $p, p^\star$ are not accessed in steps $\{i+1, \ldots, j\}$, OPT has neither $p$ nor $p^\star$ in its cache at the end of step $j$. Hence we get a feasible solution OPT$'[\sigma]$ from OPT$[\sigma]$ by evicting $p$ instead of $p^\star$ in step $i$, evicting $p^\star$ instead of $p$ in step $j$ and otherwise copying OPT's decisions. Since we did not cause additional page faults, we have OPT$'(\sigma) \leq$ OPT$(\sigma)$, hence OPT$'[\sigma]$ is still optimal.

**Case 2: $p$ is not evicted by OPT in steps $i, \ldots, j^\star$ and $j^\star = n$.**
Since $p, p^\star$ are not accessed anymore after step $i$, it makes no difference whether we evict $p$ or $p^\star$ in step $i$, and we can simply obtain OPT$'[\sigma]$ from OPT$[\sigma]$ by evicting $p$ instead of $p^\star$ in step $i$.

**Case 3: $p$ is not evicted by OPT in steps $i, \ldots, j^\star$ and $j^\star < n$.**
Then, in step $j^\star + 1$, page $p^\star$ is requested and, because OPT is demand paging, page $p$ is in OPT's cache but $p^\star$ is not. Let $p'$ be the page evicted by OPT in step $j^\star + 1$ to make room for $p^\star$. If $p' = p$, then, after step $j^\star + 1$, OPT has $p^\star$ in its cache, but not $p$. If we evict page $p$ instead of $p^\star$ in step $i$ and otherwise copy OPT's behavior, we have the same cache contents after step $j^\star + 1$ and one less page fault, which contradicts optimality of OPT. If $p' \neq p$, then, after step $j^\star + 1$, OPT has both $p^\star$ and $p$ in its cache. We construct the solution OPT$'[\sigma]$ by evicting $p$ instead of $p^\star$ in step $i$ and loading $p$ instead of $p^\star$ in step $j^\star + 1$. This solution is feasible (since it has the same cache contents after step $j^\star + 1$) and does not cause additional page faults, i.e., OPT$'(\sigma) =$ OPT$(\sigma)$. Note that OPT$'[\sigma]$ is not demand paging, but can be modified to a demand paging solution without affecting its behavior in the first $i$ steps, and without affecting its cost (as described in Remark 2.1). □

## 2.2 Lower bound for online paging

To derive a lower bound on the competitive ratio of any algorithm, we need to construct an adversarial request sequence $\sigma \in P^n$ for a given online algorithm ALG. A natural candidate is the request sequence that, in every step, requests an item that is not in ALG's cache. For such a sequence, we trivially have ALG$(\sigma) = n$. To get a bound on the competitive ratio of ALG, we need a sufficiently strong upper bound on the cost of the optimum solution. By Theorem 2.2, we can get such a bound by analyzing the performance of LFD. Observe that our construction works even if $|P| = k + 1$.

**Lemma 2.3.** *The cost of* LFD *for instances $\sigma \in P^n$ with $|P| = k + 1$ is bounded by*

$$\text{LFD}(\sigma) \leq \lceil n/k \rceil .$$

*Proof.* Assume that LFD has a page fault upon request $r_i \in \sigma$ and evicts page $p \in P$. Since $|P| = k + 1$ and the cache starts with $k$ elements, the next page fault occurs when $p$ is requested the next time. By definition of LFD, all other $k - 1$ pages except $r_i$ in the cache are requested before this happens. Hence, LFD has a page fault only at most once in every $k$ steps. □

If we choose the length $n$ of our adversarial sequence $\sigma$ to be a multiple of $k$, we immediately get $\text{ALG}(\sigma) = n \geq k \cdot \text{LFD}(\sigma) = k \cdot \text{OPT}(\sigma)$. Since our construction works for any online algorithm, this bound holds in general. For $n \to \infty$, we can use Proposition 1.12 to obtain the following bound.

**Theorem 2.4** (Sleator and Tarjan [1985])**.** *The competitive ratio of any deterministic paging algorithm is at least $k$, even when $|P| = k + 1$.*

*Proof.* For the sake of contradiction, let ALG be a $\rho$-competitive paging algorithm with $\rho < k$ for instances with $|P| = k + 1$. Let $\sigma^{(i)} \in P^n$ be the request sequence of length $n = ik$ that always requests the unique page not in ALG's cache. Then, $\lim_{i\to\infty} \text{ALG}(\sigma^{(i)}) = \infty$. By Lemma 2.3 and Theorem 2.2, we have $\text{ALG}(\sigma^{(i)}) = n \geq k \cdot \text{LFD}(\sigma^{(i)}) = k \cdot \text{OPT}(\sigma^{(i)})$. It follows that $\limsup_{i\to\infty} \frac{\text{ALG}(\sigma^{(i)})}{\text{OPT}(\sigma^{(i)})} \geq k > \rho$. By Proposition 1.12, this is a contradiction with ALG being $\rho$-competitive. □

## 2.3 Online paging algorithms

Now that we have established an understanding of the offline optimal algorithm LFD and have a bound on the best possible competitive ratio, we shift our attention to finding good online algorithms. Since we may restrict ourselves to demand paging algorithms, possible candidates only need to differ in the decision which page to evict upon a page fault. We list a few canonical heuristics that respond differently to page faults:

**Least-Recently-Used (LRU):** evict the page that was least recently accessed

**First-In-First-Out (FIFO):** evict the page that has been in the cache the longest since it was loaded (or any page in the cache initially)

**Last-In-First-Out (LIFO):** evict the page that was most recently loaded into the cache

**Least-Frequently-Used (LFU):** evict any page that has been least used overall

**Flush-When-Full (FWF):** at the start and upon a page fault with a full cache, evict all pages (this is not demand paging)

We start by analyzing FIFO.

**Theorem 2.5** (Sleator and Tarjan [1985])**.** FIFO *is strictly $k$-competitive for the paging problem.*

*Proof.* Fix a request sequence $\sigma \in P^n$ for which FIFO has a page fault for some request $r_i = p$. By definition, before FIFO can have another page fault caused by another access $r_j = p$ with $j > i$, all $k$ pages that were in the cache at the start of step $i$ must have been evicted as well as $p$. This means that there must be $k + 2$ total page faults in steps $i$ through $j$, i.e., $j \geq i + k + 1$. Overall, we get that any set of $k + 1$ consecutive page faults must be caused by distinct elements.

We partition the request sequence $\sigma$ into phases $\pi_1, \ldots, \pi_N$, such that $\sigma = \pi_1 \oplus \cdots \oplus \pi_N$, FIFO incurs at most $k$ page faults in each phase, and $(|\pi_N|, \ldots, |\pi_2|, |\pi_1|)$ is lexicographically maximized (see Figure 3). Note that, by definition, FIFO incurs exactly $k$ page faults in all phases except $\pi_1$, and at most $k$ page faults in phase $\pi_1$. We get $\text{FIFO}(\sigma) \leq Nk$.

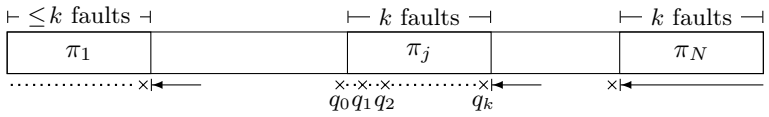


Figure 3: Illustration of the partitioning of the request sequence $\sigma$ in the proof of Theorem 2.5.

Consider any phase $\pi_j \neq \pi_1$, let $q_1, \ldots, q_k$ be the pages that caused faults during the phase, and let $q_0$ be the last page that was requested before phase $\pi_j$ (i.e., the last page requested in phase $\pi_{j-1}$). Recall that $\sigma$ was chosen such that FIFO has at least one page fault. Thus, by definition, phase $\pi_1$ contains at least one page fault, since otherwise $(\pi_1 \oplus \pi_2), \pi_3, \ldots, \pi_N$ would be a lexicographically larger phase partitioning (with respect to $(|\pi_N|, \ldots, |\pi_3|, |\pi_1 \oplus \pi_2|)$). Similarly, the access to $q_0$ just before phase $\pi_j$ must have caused a page fault, otherwise $q_0$ could be included in phase $\pi_j$ instead. Since page faults can only repeat after $k+1$ distinct requests, we know that $q_0, \ldots, q_k$ must all be distinct. Since $q_0$ must be in OPT's cache after it was accessed, $q_0$ is in OPT's cache at the end of phase $\pi_{j-1}$. Since $k$ pages other than $q_0$ are accessed during phase $\pi_j$, OPT has at least one page fault during $\pi_j$. Since phase $\pi_1$ causes at least one page fault for FIFO, there must be at least one page in $\pi_1$ that was not in the initial cache $C_0$. But this means that OPT also has at least one page fault in phase $\pi_1$. We get $\text{OPT}(\sigma) \geq N$, and thus

$$\text{FIFO}(\sigma) \leq k \cdot \text{OPT}(\sigma). \qquad \square$$

**Corollary 2.6.** *The paging problem with cache size $k$ has (strict) competitive ratio exactly $k$.*

## 2.4 Phase partitioning

Motivated by the proof of Theorem 2.5 we want to establish a proof scheme that can be applied more generally. The main idea behind the proof of Theorem 2.5 was to partition the request sequence $\sigma$ into phases, with the property that each phase causes at least one page fault for OPT. The partitioning of Theorem 2.5 depended on FIFO's behavior, but we can define an even simpler partitioning with the same property and independent of a specific algorithm.

**Definition 2.7.** The *$k$-phase partitioning* of a request sequence $\sigma$ is the unique partition $\sigma = \pi_1 \oplus \pi_2 \oplus \cdots \oplus \pi_N$ for which each *phase* $\pi_j$ contains at most $k$ distinct requests and $(|\pi_1|, |\pi_2|, \ldots, |\pi_N|)$ is maximized lexicographically.

For example, consider $\sigma = (1, 2, 4, 2, 1, 3, 5, 2, 5, 5, 3, 3, 3, 1, 2, 3, 4)$. Then the 3-phase partitioning is $\pi_1 = (1, 2, 4, 2, 1)$, $\pi_2 = (3, 5, 2, 5, 5, 3, 3, 3)$, $\pi_3 = (1, 2, 3)$, $\pi_4 = (4)$. Note that the partitioning is indeed independent of any algorithm.

In order to prove that a given algorithm ALG is $k$-competitive as in Theorem 2.5, we need that ALG does not have more than $k$ page faults per phase. This is obviously the case if for every phase $\pi_j$ it is true that no pages $p \in \pi_j$ are evicted during phase $\pi_j$. The following definition gives a slightly weaker sufficient condition in terms of a marking scheme.

**Definition 2.8.** A *marking algorithm* is an algorithm that never evicts a marked page, where a page is considered marked if it has previously been requested during the current phase of the $k$-phase partitioning.

Note that FIFO itself no longer falls under the simplified definition of marking algorithms. To see this, consider the partitioned request sequence $\sigma = \pi_1 \oplus \pi_2 = (1, 2, 3) \oplus (4, 2, 1)$ for $k = 3$ and $C_0 = \{4, 5, 6\}$. Upon the last access to 1, the marked page 2 will be evicted by FIFO. On the other hand, FWF obviously is a marking algorithm, since it only evicts pages once a new phase starts.

**Proposition 2.9.** FIFO *is not a marking algorithm.*

We can now carry over the idea from the proof of Theorem 2.5 to marking algorithms.

**Theorem 2.10** (Torng [1998]). *Every marking algorithm is $k$-competitive for the paging problem with (constant) cache size $k$.*

*Proof.* Let ALG be any marking algorithm and consider any input $\sigma \in P^n$ with $k$-phase partitioning $\pi_1, \ldots, \pi_N$. Since ALG is a marking algorithm, it has at most one page fault per distinct element requested in every phase. By definition, every phase contains at most $k$ distinct requests, hence $\text{ALG}(\sigma) \leq Nk$.

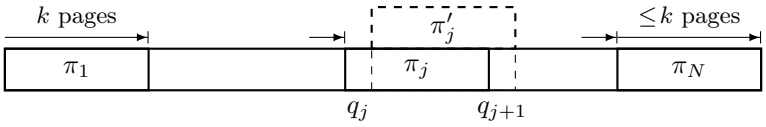


Figure 4: Illustration of the partitioning of the request sequence $\sigma$ in the proof of Theorem 2.10.

On the other hand, consider any phase $\pi_j \neq \pi_N$ and let $q_j$, $q_{j+1}$ be the first requests of phases $\pi_j$ and $\pi_{j+1}$, respectively (see Figure 4). Since $\pi$ is lexicographically maximal, $\pi_j$ contains exactly $k$ distinct requests and $q_{j+1} \notin \pi_j$. Let $\pi'_j$ be the shifted phase starting with the second request of $\pi_j$ and ending with the first request of $\pi_{j+1}$, i.e., $(q_j) \oplus \pi'_j := \pi_j \oplus (q_{j+1})$. Then $\pi'_j$ contains exactly $k$ requests different from $q_j$. Since OPT must have $q_j$ in its cache at the start of $\pi'_j$, it has at least one cache fault during $\pi'_j$. Since the positions of the shifted phases for all phases $\pi_j \neq \pi_N$ are mutually disjoint, we have $\text{OPT}(\sigma) \geq N - 1$, and, hence, $\text{ALG}(\sigma) \leq Nk \leq k \cdot \text{OPT}(\sigma) + k$. □

**Corollary 2.11** (Karlin et al. [1988]). FWF *is $k$-competitive for the paging problem with (constant) cache size $k$.*

# 3 Randomized Algorithms

In Chapter 2 we saw that no deterministic online algorithm can have a better competitive ratio for the paging problem than the cache size $k$. This lower bound is easy for the adversary to achieve simply by always requesting the exact page the algorithm just evicted. Of course this adversarial strategy relies on perfect knowledge of the behavior of the algorithm in every step. Consequently, there is hope that we can better defend against the adversary if we are less predictable, i.e., if we include random choices into the behavior of our algorithm. Formally, we can express this by defining randomized online algorithms that, for every request sequence, randomly choose an algorithm from the set of all deterministic algorithms $\mathcal{A}_{\text{det}}$ according to some distribution, and then deterministically execute the chosen algorithm. Note that, on any finite set of instances, every online algorithm that involves random choices can be expressed in this way (see Isbell [1957]).

**Definition 3.1.** A *randomized online algorithm* $\text{ALG}_p$ is identified with a probability distribution $p\colon \mathcal{A}_{\text{det}} \to [0,1]$ with $\sum_{\text{ALG}\in\mathcal{A}_{\text{det}}} p(\text{ALG}) = 1$.

In order to define the competitive ratio of a randomized online algorithm $\text{ALG}_p$, we need to specify whether or not the adversary may use any knowledge about the outcome of the random choice that $\text{ALG}_p$ makes at the start of each run to come up with an adversarial input sequence. Of course, if we allow the adversary to be adaptive to the random choice of $\text{ALG}_p$, then the same lower bound construction for paging still works. In this lecture, we will restrict ourselves to an *oblivious adversary* that must specify a worst-case instance without any knowledge regarding $\text{ALG}_p$'s random choice.

**Definition 3.2.** A randomized online algorithm $\text{ALG}_p$ is *$\rho$-competitive* if there is a constant $\alpha \geq 0$, such that, for all $\sigma \in \Sigma$,

$$\mathbb{E}[\text{ALG}_p(\sigma)] := \mathbb{E}_{\text{ALG}\sim p}[\text{ALG}(\sigma)] \leq \rho \cdot \text{OPT}(\sigma) + \alpha.$$

$\text{ALG}_p$ is *strictly $\rho$-competitive,* if the above holds for $\alpha = 0$.

**Definition 3.3.** The *(strict) randomized competitive ratios* of an algorithm and a problem are defined analogously to the deterministic case (Definitions 1.10 and 1.11).

## 3.1 Randomized paging algorithms

We can easily define a "randomized algorithm" RAND that prevents the adversarial strategy of requesting the previously evicted page by evicting a page uniformly at random upon each page fault.[1] However, this algorithm does not beat the deterministic bound on the competitive ratio.

**Proposition 3.4.** *The competitive ratio of* RAND *is at least $k$, even when $|P| = k+1$.*

*Proof.* We use $P = \{p_0, p_1, \ldots, p_k\}$, $C_0 = \{p_1, \ldots, p_k\}$, and we define the infinite sequence $\sigma = (p_0, p_1, \ldots, p_{k-1}, p_0, \ldots)$. We consider the request sequence $\sigma_{\leq n}$ for $n \in \mathbb{N}$. For any such sequence, OPT evicts $p_k$ in the first step and $\text{OPT}(\sigma_{\leq n}) = 1$. On the other hand, RAND continues to have page faults until it evicts $p_k$, assuming $n$ is large enough. Since RAND evicts pages uniformly at random, its probability to evict $p_k$ upon a page fault is $1/k$. Hence, the expected number of page faults until $p_k$ is evicted approaches $k$ as $n$ goes to infinity. We have $\lim_{n\to\infty} \mathbb{E}[\text{RAND}(\sigma_{\leq n})] = k \cdot \text{OPT}(\sigma_{\leq n})$.

Since $\mathbb{E}[\text{RAND}(\sigma_{\leq n})]$ is bounded independently of $n$, the above only proves that RAND does not have a *strict* competitive ratio smaller than $k$ (as in Proposition 1.12). Let $\sigma^{(j)} := (p_j, p_{j+1} \ldots, p_{j-2}, p_j, \ldots)$ with $j \in \{0, \ldots, k\}$ and all indices modulo $k+1$ from now on. Then $\sigma^{(0)}$ is the sequence

[1] Note that RAND is not formally a randomized algorithm.

we defined above (since $-2 \mod (k+1) \equiv k-1$) and in $\sigma^{(j)}$ the role of $p_k$ in $\sigma^{(0)}$ is played by $p_{j-1}$. We can make the cost of OPT grow unboundedly, by defining the request sequence $\sigma^{(n,m)} := \sigma^{(0)}_{\leq n} \oplus \sigma^{(k)}_{\leq n} \oplus \sigma^{(k-1)}_{\leq n} \oplus \cdots \in P^{mn}$ consisting of $m \in \mathbb{N}$ phases of the form $\sigma^{(j)}_{\leq n}$ for $j \in \{0, \ldots, k\}$. As above, OPT has at most one page fault per phase, and hence $\text{OPT}(\sigma^{(n,m)}) \leq m$. If we let $n$ go to infinity for a fixed value of $m$, then the probability that RAND does not have page $p_{j-1}$ in its cache after a phase of the form $\sigma^{(j)}_{\leq n}$ approaches one. Hence, the number of page faults of RAND approaches $k$ per phase, as above.

Now if RAND is $\rho$-competitive, then there is a constant $\alpha \geq 0$ such that, for all $n, m \in \mathbb{N}$, we have $\mathbb{E}[\text{RAND}(\sigma^{(n,m)})] \leq \rho \cdot \text{OPT}(\sigma^{(n,m)}) + \alpha$. It follows that (we let $n$ grow much faster than $m$)

$$\rho \geq \lim_{m\to\infty} \lim_{n\to\infty} \frac{\mathbb{E}[\text{RAND}(\sigma^{(n,m)})] - \alpha}{\text{OPT}(\sigma^{(n,m)})} \geq \lim_{m\to\infty} \frac{km - \alpha}{m} = k.$$

□

We can summarize the approach used in the proof of Proposition 3.4 to obtain a lower bound on the randomized competitive ratio analogously to Proposition 1.12. Note that it would be suffcient to require $\lim_{i\to\infty} \mathbb{E}[\text{ALG}_p(\sigma^{(i)})] = \infty$ below.

**Proposition 3.5.** *Let $(\sigma^{(i)} \in \Sigma)_{i\in\mathbb{N}}$ be a sequence of request sequences with*

$$\lim_{i\to\infty} \text{OPT}(\sigma^{(i)}) = \infty.$$

*Then, every randomized online algorithm $\text{ALG}_p$ has a competitive ratio of at least*

$$\limsup_{i\to\infty} \frac{\mathbb{E}_{\text{ALG}\sim p}[\text{ALG}(\sigma^{(i)})]}{\text{OPT}(\sigma^{(i)})}.$$

Note that the $k$-phase partitioning of the request sequence $\sigma$ defined in the beginning of the proof of Proposition 3.4 consists of a single phase. The problem with RAND on this instance is that it evicts pages that were previously requested during the phase, i.e., its problem is that it is not a marking algorithm. We turn RAND into a randomized marking algorithm RMARK, by marking pages after they have been requested, unmarking all pages at the beginning of a phase, and evicting an *unmarked* page uniformly at random upon each page fault. It turns out that this modification suffices to perform much better than the best deterministic online algorithm.

**Theorem 3.6** (Fiat et al. [1991])**.** RMARK *is strictly $2\mathcal{H}_k$-competitive, where $\mathcal{H}_k := \sum_{i=1}^{k} \frac{1}{i} \approx \ln k$ is the $k$-th harmonic number.*

*Proof.* Fix a request sequence $\sigma \in P^n$ and let its $k$-phase partitioning be $\pi_1 \oplus \pi_2 \oplus \cdots \oplus \pi_N = \sigma$. Consider any phase $\pi_i$, call the pages in the cache at the start of the phase *old* and all other pages *new*. Let $m_i$ denote the number of (distinct) new pages requested in phase $\pi_i$. Obviously, each new page causes a page fault for RMARK. We compute the expected number of page faults in phase $\pi_i$ caused by old pages. Let $r \in \pi_i$ be the $j$-th (distinct) old page that is requested during phase $\pi_i$. At the point where $r$ is requested for the first time during $\pi_i$, there are at least $k - m_i$ old pages in the cache, and exactly $j-1$ of them are marked, hence there are at least $k - m_i - (j-1)$ unmarked old pages in the cache. Also, the total number of unmarked old pages, both in and not in the cache, is exactly $k - (j-1)$. Overall, the probability that page $r$ causes a page fault is at most

$$1 - \frac{k - m_i - (j-1)}{k - (j-1)} = \frac{m_i}{k - j + 1}.$$

Since at most $k - m_i$ old pages are accessed in phase $\pi_i$ (fewer than $k - m_i$ only in the last phase), the expected number of page faults of RMark in phase $\pi_i$ is at most

$$\begin{aligned} m_i + \sum_{j=1}^{k-m_i} \frac{m_i}{k-j+1} &= m_i \cdot \left(1 + \frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{m_i+1}\right) \\ &= m_i \cdot (1 + \mathcal{H}_k - \mathcal{H}_{m_i}) \\ &\leq m_i \mathcal{H}_k. \end{aligned}$$

The total expected cost of RMark becomes

$$\mathbb{E}[\text{RMark}(\sigma)] \leq \sum_{i=1}^{N} m_i \mathcal{H}_k.$$

It remains to show a lower bound on $\text{Opt}(\sigma)$. By definition, RMark has all pages requested in a phase in the cache at the end of the phase. Since exactly $k$ distinct pages are requested in every phase except the last, we have that the number of distinct pages accessed in the two consecutive phases $\pi_{i-1}, \pi_i$, $i > 1$, is exactly $k + m_i$. It follows that Opt has at least $m_i$ page faults in phases $\pi_{i-1}$ and $\pi_i$ combined. Also Opt has at least $m_1$ page faults in the first phase. We get a bound for Opt by separately considering even and odd values of $i$:

$$\text{Opt}(\sigma) \geq \max\left\{ \sum_{l=1}^{\lfloor N/2 \rfloor} m_{2l}, \sum_{l=0}^{\lfloor (N-1)/2 \rfloor} m_{2l+1} \right\} \geq \frac{1}{2} \sum_{l=1}^{\lfloor N/2 \rfloor} m_{2l} + \frac{1}{2} \sum_{l=0}^{\lfloor (N-1)/2 \rfloor} m_{2l+1} = \frac{1}{2} \sum_{i=1}^{N} m_i.$$

Finally, we conclude that

$$\mathbb{E}[\text{RMark}(\sigma)] \leq \mathcal{H}_k \sum_{i=1}^{N} m_i \leq 2\mathcal{H}_k \cdot \text{Opt}(\sigma).$$

□

## 3.2 Yao's principle

So far, we derived lower bounds for the competitive ratios of online algorithms by taking the perspective of an adversary and constructing a request sequence tailored to a given algorithm, in such a way that we respond to every possible action of the algorithm in the worst possible manner. This is no longer possible for randomized algorithms, since we do not know the outcomes of the algorithm's random choices beforehand. We now have to construct a request sequence for which the algorithm performs badly in expectation over all possible combinations of random choices the algorithm makes, as stated by Proposition 3.5. In general, this is a daunting task.

Yao's principle provides a crucial tool to prove lower bounds by allowing us to restrict ourselves to deterministic algorithms. In a way, Yao's principle allows to trade the randomness in the algorithm for randomness in the input sequence. Instead of designing a singe sequence of request sequences with high expected cost for *any fixed randomized* algorithm, we can design a sequence of *distributions* of input sequences with high expected cost for *every deterministic* algorithm. This turns out to be much more manageable in many cases.

**Theorem 3.7** (Yao [1977])**.** *Let* $(q_i : \Sigma \to [0,1])_{i \in \mathbb{N}}$ *be a sequence of probability distributions with*

$$\lim_{i \to \infty} \mathbb{E}_{\sigma \sim q_i}[\text{Opt}(\sigma)] = \infty.$$

*Then, every randomized online algorithm has a competitive ratio of at least*

$$\limsup_{i \to \infty} \frac{\inf_{\text{Alg} \in \mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma \sim q_i}[\text{Alg}(\sigma)]}{\mathbb{E}_{\sigma \sim q_i}[\text{Opt}(\sigma)]}.$$

*Proof.* Let $\text{ALG}_p$ be the randomized algorithm associated with the probability distribution $p\colon \mathcal{A}_{\text{det}} \to [0,1]$. Assume that $\text{ALG}_p$ has competitive ratio $\rho$, i.e., there is a constant $\alpha \geq 0$ such that, for all $\sigma \in \Sigma$,

$$\mathbb{E}_{\text{ALG}\sim p}[\text{ALG}(\sigma)] \leq \rho \cdot \text{OPT}(\sigma) + \alpha.$$

In particular, for every distribution $q_i$, $i \in \mathbb{N}$ of the given sequence, we obtain

$$\begin{aligned}\mathbb{E}_{\sigma\sim q_i}[\mathbb{E}_{\text{ALG}\sim p}[\text{ALG}(\sigma)]] &= \sum_{\sigma\in\Sigma} q_i(\sigma)\cdot \mathbb{E}_{\text{ALG}\sim p}[\text{ALG}(\sigma)] \\ &\leq \rho \cdot \sum_{\sigma\in\Sigma} q_i(\sigma)\cdot \text{OPT}(\sigma) + \alpha \cdot \sum_{\sigma\in\Sigma} q_i(\sigma) \\ &= \rho \cdot \mathbb{E}_{\sigma\sim q_i}[\text{OPT}(\sigma)] + \alpha.\end{aligned}$$

Since $p$ and $q_i$ are independent, the corresponding expectations commute. We get

$$\begin{aligned}\rho \cdot \mathbb{E}_{\sigma\sim q_i}[\text{OPT}(\sigma)] + \alpha &\geq \mathbb{E}_{\sigma\sim q_i}[\mathbb{E}_{\text{ALG}\sim p}[\text{ALG}(\sigma)]] \\ &= \mathbb{E}_{\text{ALG}\sim p}[\mathbb{E}_{\sigma\sim q_i}[\text{ALG}(\sigma)]] \\ &\geq \inf_{\text{ALG}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_i}[\text{ALG}(\sigma)],\end{aligned}$$

for every $i \in \mathbb{N}$. Solving for $\rho$ and taking lim sup gives

$$\begin{aligned}\rho &\geq \limsup_{i\to\infty} \left[\frac{\inf_{\text{ALG}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_i}[\text{ALG}(\sigma)]}{\mathbb{E}_{\sigma\sim q_i}[\text{OPT}(\sigma)]} - \frac{\alpha}{\mathbb{E}_{\sigma\sim q_i}[\text{OPT}(\sigma)]}\right] \\ &\geq \limsup_{i\to\infty} \frac{\inf_{\text{ALG}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_i}[\text{ALG}(\sigma)]}{\mathbb{E}_{\sigma\sim q_i}[\text{OPT}(\sigma)]} - \limsup_{i\to\infty} \frac{\alpha}{\mathbb{E}_{\sigma\sim q_i}[\text{OPT}(\sigma)]} \\ &= \limsup_{i\to\infty} \frac{\inf_{\text{ALG}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_i}[\text{ALG}(\sigma)]}{\mathbb{E}_{\sigma\sim q_i}[\text{OPT}(\sigma)]},\end{aligned}$$

where we used $\lim_{i\to\infty} \mathbb{E}_{\sigma\sim q_i}[\text{OPT}(\sigma)] = \infty$. □

If we only need a bound for the *strict* competitive ratio, we can use a simplified version of Yao's principle (proof: exercise).

**Corollary 3.8.** *Let $q\colon \Sigma \to [0,1]$ be any probability distribution. Then, every randomized online algorithm has a strict competitive ratio of at least*

$$\frac{\inf_{\text{ALG}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q}[\text{ALG}(\sigma)]}{\mathbb{E}_{\sigma\sim q}[\text{OPT}(\sigma)]}.$$

*Remark* 3.9. Note that Yao's principle is tight in the sense that we can always find a distribution to prove any valid lower bound on the competitive ratio.

## 3.3 Lower bound for randomized paging

Equipped with Yao's principle we are now able to derive an (almost) tight lower bound on the randomized competitive ratio of the paging problem.

**Theorem 3.10** (Fiat et al. [1991])**.** *The competitive ratio of any randomized paging algorithm is at least $\mathcal{H}_k$, even when $|P| = k+1$.*

*Proof.* We fix $P = \{p_0, \ldots, p_k\}$ and the initial cache $C_0 = \{p_1, \ldots, p_k\}$. We want to apply Yao's principle (Theorem 3.7), i.e., we need to provide a sequence $(q_n)_{n\in\mathbb{N}}$ of probability distributions over input instances. We describe $q_n$ by a randomized strategy to construct a request sequence $\sigma = (r_1, \ldots, r_{n'})$ of length $n' \geq n$: We set $r_1 = p_0$ and iteratively choose $r_i$, $i = 2, 3, \ldots$ uniformly at random from $P \setminus \{r_{i-1}\}$, until $i > n$ and the last phase of the $k$-phase partitioning of $(r_1, \ldots, r_i)$

consists of exactly one request. At this point, we terminate with $\sigma = (r_1, \ldots, r_{i-1})$. We let $q_n(\sigma)$ be the probability that sequence $\sigma$ is produced by this procedure.

Consider the $k$-phase partitioning $\sigma = \pi_1 \oplus \pi_2 \oplus \cdots \oplus \pi_N$ of any sequence $\sigma = (r_1, \ldots, r_{n'}) \in P^{n'}$ with $q_n(\sigma) > 0$. Let $z_j$ denote the first request during phase $\pi_j$, let $\pi'_1 = (z_1)$ and let $\pi'_{j\geq 2}$ be defined via $(z_{j-1}) \oplus \pi'_j = \pi_{j-1} \oplus (z_j)$. Observe that $\sigma = \pi'_1 \oplus \cdots \oplus \pi'_N$. Since $r_1 \notin C_0$ and since $z_{j-1}$must be in the cache at the start of phase $\pi'_{j\geq 2}$, every offline algorithm must have a page fault during $\pi'_j$ for every $j \in \{1, \ldots, N\}$. On the other hand, there is exactly one page that is not requested in each phase, therefore the offline algorithm that evicts this page upon the first page fault in a phase never has more than one page fault per phase. Hence, OPT has exactly $N$ page faults.

Let $X$ be the random variable that measures the length of a phase $\pi_j$. Note that, by construction, $\mathbb{E}_{\sigma\sim q_n}[X]$ is independent of $j$. We claim that $\mathbb{E}_{\sigma\sim q_n}[X] = k\mathcal{H}_k$. With this, and since OPT has one fault per phase, we immediately get

$$\lim_{n\to\infty} \mathbb{E}_{\sigma\sim q_n}[\textsc{Opt}(\sigma)] = \lim_{n\to\infty} \mathbb{E}_{\sigma\sim q_n}[N] = \infty,$$

as required by Theorem 3.7.

Now, consider the behavior of *any* deterministic online (demand paging) algorithm ALG for request sequence $\sigma$. At the beginning of step $i \geq 2$, the cache of ALG contains all but one page $r \in P \setminus C$ and the probability that $r$ is the next request is exactly $1/k$ by construction of $q_n$, because $r_i \in P \setminus \{r_{i-1}\}$ and $r_{i-1} \in C$. The first request $r_1$ always causes a page fault. Hence, the expected number of page faults of ALG per phase is at least

$$\frac{1}{k} \sum_{l=1}^{\infty} \Pr[X \geq l] = \mathbb{E}_{\sigma\sim q_n}[X]/k.$$

Since this holds for every choice of ALG, and since OPT never has more than one fault per phase, we get

$$\limsup_{n\to\infty} \frac{\inf_{\textsc{Alg}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_n}[\textsc{Alg}(\sigma)]}{\mathbb{E}_{\sigma\sim q_n}[\textsc{Opt}(\sigma)]} \geq \frac{\mathbb{E}[X]}{k} = \mathcal{H}_k,$$

and we can apply Theorem 3.7 to complete the proof.

It remains to prove our claim that $\mathbb{E}_{\sigma\sim q_n}[X] = k\mathcal{H}_k$. Note that for every request after the first, we draw uniformly at random from $k$ possibilities, and the next phase starts once we have drawn all pages. To compute the expected length of a phase, we need to solve the *coupon collector problem*: "If we draw coupons from an urn with replacement, how many do we have to draw to get each at least once?" Let $X_j$ be the random variable that counts the number of requests during a phase until a new page is requested, once $j-1$ different pages have already been requested. Then $X = \sum_{j=1}^{k+1} X_j - 1$. By definition, $X_1 = 1$. Consider $X_j$ with $j \geq 2$. If $j-1$ pages have been requested before some request $r_i$, the probability that $r_i$ is again one of these pages is $(j-2)/k$, since $r_i$ is chosen uniformly at random from $P \setminus \{r_{i-1}\}$. Therefore, $X_j$ is a geometric random variable with success probability $s_j = 1 - \frac{j-2}{k}$, and

$$\mathbb{E}_{\sigma\sim q_n}[X_{j\geq 2}] = \frac{1}{s_j} = \frac{k}{k-j+2}.$$

By linearity of expectation, we get

$$\begin{aligned}
\mathbb{E}_{\sigma\sim q_n}[X] &= \sum_{j=1}^{k+1} \mathbb{E}_{\sigma\sim q_n}[X_j] - 1 \\
&= \sum_{j=2}^{k+1} \frac{k}{k-j+2} \\
&= k \sum_{j=1}^{k} \frac{1}{j} = k\mathcal{H}_k,
\end{aligned}$$

which completes the proof. □

*Remark* 3.11. An $\mathcal{H}_k$-competitive randomized algorithm was given by McGeoch and Sleator [1991].

# 4 The List Update Problem

In the previous chapter, we considered the abstract problem of dynamically reorganizing a basic random access data structure that allows fast access to an arbitrary subset of its data. We now introduce a similar problem with an unsorted linked list as our underlying data structure. Essentially, a linked list is managed by keeping a reference to the first element of the list and equipping each element with a reference to its successor in the list. To access an element at position $i$ of a linked list, we need to traverse the first $i-1$ elements, i.e., accessing items further down the list is more expensive than accessing elements near the front. After extracting an element it may therefore make sense to return it in a position closer to the front in order to speed up further accesses to the same element. To simplify our analysis we also allow to invest extra time to switch the order of any consecutive items. In the standard form, the list update problem is given as follows.[2]

| List Update Problem | |
|---|---|
| **given:** | set of (different) items $X$, $l := \|X\|$, stored in a list $L_0$ |
| **online:** | sequence $\sigma = (x_1, \ldots, x_n) \in X^n$ of items accessed in this order |
| **actions** (step $i$)**:** | transpose any number of consecutive items (**paid**) |
| | move $x_i$ closer to the front of the list (**free**) |
| **objective:** | minimize the total cost $\sum_{i=1}^{n} \textsc{Alg}_i(\sigma) = \sum_{i=1}^{n}(s_i + p_i)$ |
| | $s_i$: number of items in front of $x_i$ at the beginning of step $i$ |
| | $p_i$: number of paid actions in step $i$ |

On first glance, it seems hopeless to design a good online algorithm for this problem: How can we know whether an element will be accessed often in the future and should be moved closer to the front? This is where the distinction between measuring *expected* or measuring *worst-case* performance makes all the difference. Of course, if we try to optimize expected cost with respect to a uniform distribution of accesses to elements, there is nothing to be done, since every strategy (that does not needlessly transpose items) has the same quality in expectation. However, in competitive analysis we have to defend against the worst case! For example, if we choose not to move elements at all, the adversary has the obvious strategy of repeatedly accessing the last element of the list. The offline optimum for this instance is very cheap, because it immediately moves the last element to the front, while our algorithm needs to repeatedly traverse the entire list and therefore incurs high cost. An obvious way to prevent this worst-case is to use the *move-to-front* heuristic Mtf: In every step, move the accessed element to the first position of the list. To analyze this simple algorithm, we first need to introduce a very important technique of competitive analysis: the *potential function method* (or *amortized analysis*).

## 4.1 Potential function method (amortized analysis)

The competitive analysis of an online algorithm Alg is easiest if we can bound the cost $\textsc{Alg}_i(\sigma)$ in each step individually by the corresponding cost $\textsc{Opt}_i(\sigma)$ of the offline optimum. Often however, it is impossible to guarantee that in *every* step Alg incurs a relatively small cost. In the Section 2.4, we introduced the phase partitioning technique in order to bound the costs for different phases separately. But even if we are unable to bound Alg's cost per step/phase, the algorithm may still be competitive if expensive steps only occur rarely and only if there are sufficiently many cheap steps to compensate. For example, Mtf may have expensive steps in which elements are accessed that Opt has near the front of the list while they are near the back in Alg's list. However, for any element near the front to again reach the back of the list, many other elements need to be accessed by Mtf, i.e., there must have been many cheaper steps (see Figure 5). We can try to account for

[2] We use a cost model where accessing the element at position $i$ incurs cost $i-1$; this is often referred to as the *reduced cost model* in the literature.

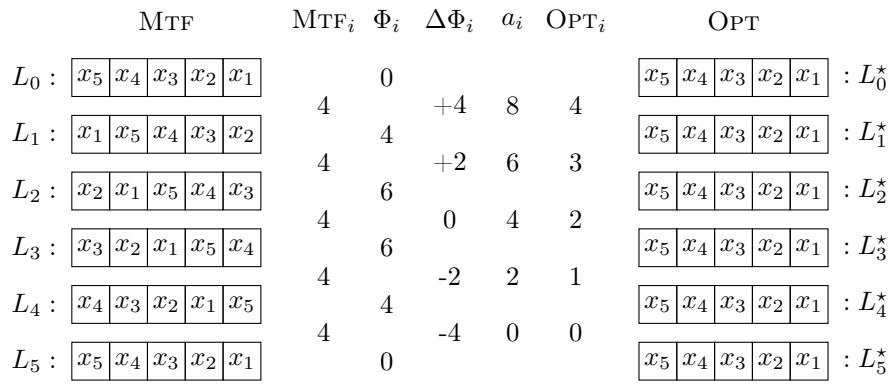


Figure 5: A bad example for MTF for $\sigma = (x_1, x_2, x_3, x_4, x_5)$. While it is possible to force $\text{MTF}_i(\sigma) = n - 1 = 4$ and $\text{OPT}_i(\sigma) = 0$, this only happens after $n - 1 = 4$ cheaper steps. Note that after the expensive step both lists become more similar.

this effect by comparing the total cost of ALG with the total cost of OPT, but this is often difficult, because we need to consider any possible sequence of requests.

The principal idea of amortized analysis is to charge cheap steps a little extra cost in order to pay for eventual expensive steps. To understand this, we first consider an intuitive example: Take a student Charlie that still has her parents paying for her meals at the university. Because Charlie does not want to waste her parents' money, she tries to spend as little as possible in the cafeteria. Say she spends only 2€ for lunch each day, but after every ten consecutive days of cafeteria she needs a break and goes to a nice restaurant that costs 13€. Sometime in the middle of the semester, Charlie's parents confront her about seeing her at the restaurant, which gives them the impression that Charlie is abusing their support. Now Charlie needs an easy argument to convince her parents that she is effectively only spending a reasonable amount of at most 3€ a day. Strictly speaking, of course, this is not the case. However, it is the case in an *amortized* sense: At *every* moment of the semester it is correct that her *average* spending is at most 3€ per lunch. To explain this easily, Charlie uses a potential function argument: She has a piggy bank at home (her *potential function* $\Phi$), and every day she takes 3€ from her parents' money, puts 1€ into the bank, and uses the remaining 2€ to pay for lunch at the cafeteria. Every eleventh day, the takes the 10€ that are in the bank plus 3€ from her parents to pay for the restaurant. With this potential function argument, her parents are quickly convinced that Charlie effectively only spends 3€ per day (i.e., her *amortized cost* $a_i$ is 3€), because she only takes 3€ from her parents money every day. We now generalize this argument formally.

**Definition 4.1.** A *potential function* is a function $\Phi\colon \mathbb{N} \times \Sigma \to \mathbb{R}$ that specifies a potential value $\Phi_i(\sigma) := \Phi(i, \sigma) = \Phi(i, \sigma_{\leq i})$ after every time step $i \in \{0, \ldots, n\}$ and for every instance $\sigma \in \Sigma$, $|\sigma| \geq i$. The *amortized cost* in step $i \in \{1, \ldots, n\}$ of an online algorithm ALG with respect to $\Phi$ and an instance $\sigma \in \mathcal{I}$ is

$$a_i(\sigma) := \text{ALG}_i(\sigma) + \Delta\Phi_i(\sigma) := \text{ALG}_i(\sigma) + \Phi_i(\sigma) - \Phi_{i-1}(\sigma).$$

In the example above, Charlie used the simple potential function $\Phi_i(\sigma) = i - \lfloor i/11 \rfloor \cdot 11$, and an amortized cost of $a_i(\sigma) = 2 + i - (i-1) = 3$ on cheap days and $a_i(\sigma) = 13 + 0 - 10 = 3$ on expensive days.

With our intuition for the potential function $\Phi$ as a piggy bank and amortized cost as the money we need to invest in each step, the following sufficient condition for competitiveness should be natural. To be slightly more general, we allow bounded negative potentials (i.e., debts) and nonzero initial potential (i.e., starting capital).

**Lemma 4.2** (potential function method)**.** *Let $\Phi$ be a potential function and let $\rho \geq 1$. If there are*

*$\alpha \geq 0$ and $\beta \in \mathbb{R}$, such that $\Phi_i(\sigma) \geq -\alpha$, $\Phi_0(\sigma) \leq \beta$ and $a_i(\sigma) \leq \rho \cdot \text{OPT}_i(\sigma)$ for all $\sigma \in \Sigma$ and $i \leq |\sigma|$, then* ALG *is $\rho$-competitive. If $\alpha = \beta = 0$, then* ALG *is strictly $\rho$-competitive.*

*Proof.* Fix any instance $\sigma = (r_1, \ldots, r_n) \in \Sigma$. Using $\text{OPT}(\sigma) = \sum_{i=1}^{n} \text{OPT}_i(\sigma)$ and $\text{ALG}(\sigma) = \sum_{i=1}^{n} \text{ALG}_i(\sigma)$, we obtain

$$\begin{aligned}
\text{ALG}(\sigma) &= \sum_{i=1}^{n} \text{ALG}_i(\sigma) \\
&= \sum_{i=1}^{n} (a_i(\sigma) - \Phi_i(\sigma) + \Phi_{i-1}(\sigma)) \\
&= \Phi_0(\sigma) - \Phi_n(\sigma) + \sum_{i=1}^{n} a_i(\sigma) \\
&\leq \beta + \alpha + \rho \cdot \sum_{i=1}^{n} \text{OPT}_i(\sigma) \\
&= \rho \cdot \text{OPT}(\sigma) + \alpha + \beta,
\end{aligned}$$

i.e., ALG is $\rho$-competitive. If $\alpha = 0$ and $\beta = 0$, we get $\text{ALG}(\sigma) \leq \rho \cdot \text{OPT}(\sigma)$, which implies that ALG is strictly $\rho$-competitive. □

Sometimes it is useful to separately consider the cost of OPT and of ALG. We can do this by rewriting the condition $a_i(\sigma) \leq \rho \cdot \text{OPT}_i(\sigma)$ of Lemma 4.2 as

$$\Delta\Phi_i(\sigma) = \Phi_i(\sigma) - \Phi_{i-1}(\sigma) \leq \rho \cdot \text{OPT}_i(\sigma) - \text{ALG}_i(\sigma). \tag{4.1}$$

This allows us to separate the contributions of OPT and ALG to the potential function. We can split each step $i$ into a substep where ALG makes its decision and one (before or after) where OPT decides. The potential function may depend on both OPT's and ALG's state/behavior (as we will see), and we can separately regard these contributions. We define $\Phi'_i$ to be the intermediate potential function value, after the first and before the second substep of step $i$. We can then define the contributions $\Delta\Phi_i^{\text{ALG}}$ and $\Delta\Phi_i^{\text{OPT}}$ of ALG and OPT to $\Delta\Phi$, respectively. If ALG acts before OPT, we define $\Delta\Phi_i^{\text{ALG}}(\sigma) := \Phi'_i(\sigma) - \Phi_{i-1}(\sigma)$ and $\Delta\Phi_i^{\text{OPT}}(\sigma) := \Phi_i(\sigma) - \Phi'_i(\sigma)$, and otherwise we exchange these definitions. In either case, $\Delta\Phi_i(\sigma) = \Delta\Phi_i^{\text{ALG}}(\sigma) + \Delta\Phi_i^{\text{OPT}}(\sigma)$, and, using eq. (4.1), we can reformulate Lemma (4.2) as the following corollary. We call this interpretation the *alternating moves approach*.

**Corollary 4.3** (potential function method, alternating moves)**.** *Let $\Phi$ be a potential function and let $\rho \geq 1$. If the following conditions hold for $\alpha \geq 0$ ($\alpha = 0$), $\beta \in \mathbb{R}$ ($\beta = 0$), and for all $\sigma \in \Sigma$ and $i \leq |\sigma|$, then* ALG *is (strictly) $\rho$-competitive:*

*(i)* $\Phi_i(\sigma) \geq -\alpha$,

*(ii)* $\Phi_0(\sigma) \leq \beta$,

*(iii)* $\Delta\Phi_i^{\text{OPT}}(\sigma) \leq \rho \cdot \text{OPT}_i(\sigma)$,

*(iv)* $\Delta\Phi_i^{\text{ALG}}(\sigma) \leq -\text{ALG}_i(\sigma)$.

## 4.2 Upper bound for move-to-front

We are now ready to analyze MTF. As discussed above (see Figure 5), MTF has an expensive access whenever an element in the back of its list is accessed that OPT has close to the front of its list, but this situation can only occur after many cheaper accesses. This characteristic suggests that we may be able to apply the potential function method.

**Theorem 4.4** (Sleator and Tarjan [1985]). MTF *is strictly 2-competitive for the list update problem.*

*Proof.* Fix a request sequence $\sigma \in \Sigma$ and let $L_i$, $L_i^\star$ denote the lists of ALG and OPT, respectively, at the end of step $i$. Further let $s_i$, $s_i^\star$ be the number of elements in front of $x_i$ in $L_{i-1}$ and $L_{i-1}^\star$, respectively, at the start of step $i$. Also, let $p_i^\star$ be the number of paid transpositions of OPT in step $i$. We then have $\text{OPT}_i(\sigma) = s_i^\star + p_i^\star$ and, since MTF does not use paid actions, we have $\text{MTF}_i(\sigma) = s_i$. We want to define a potential function that "keeps enough in the bank" to compensate for steps where OPT is significantly cheaper than MTF. We do this by keeping some budget for every inconsistency between $L$ and $L^\star$. More precisely, we set the potential $\Phi_i(\sigma)$ *after* step $i$ to be the number of inversions (flipped pairs) between $L_i$ and $L_i^\star$. Formally, we define

$$\Phi_i(\sigma) := \left|\left\{(x,y) \in X^2 \colon x \text{ before } y \text{ in } L_i \text{ but } y \text{ before } x \text{ in } L_i^\star\right\}\right|.$$

Observe that $\Phi_i(\sigma) \geq \Phi_0(\sigma) = 0$. It remains to show that this potential function does the trick, i.e., that MTF is not paying too much in cheap steps and has sufficient potential to pay for expensive steps.

In order to determine the amortized cost $a_i(\sigma)$, we need to compute the potential difference $\Delta\Phi_i(\sigma) := \Phi_i(\sigma) - \Phi_{i-1}(\sigma)$ in each step $i$. To do this, we use the alternating moves approach, where we pretend that MTF moves first and OPT only afterwards. This results in an intermediate potential value of $\Phi_i'(\sigma) := \left|\left\{(x,y) \in X^2 \colon x \text{ before } y \text{ in } L_i \text{ but } y \text{ before } x \text{ in } L_{i-1}^\star\right\}\right|$ after the move of MTF but before OPT's move. We can then separately compute $\Delta\Phi_i^{\text{MTF}}(\sigma) := \Phi_i'(\sigma) - \Phi_{i-1}(\sigma)$ and $\Delta\Phi_i^{\text{OPT}}(\sigma) := \Phi_i(\sigma) - \Phi_i'(\sigma)$ in order to obtain $\Delta\Phi_i(\sigma) = \Delta\Phi_i^{\text{MTF}}(\sigma) + \Delta\Phi_i^{\text{OPT}}(\sigma)$.

Let $X_i$ be the set of items that are in front of $x_i$ in $L_{i-1}$, but behind $x_i$ in $L_{i-1}^\star$ (indicated as "$-$" in Figure 6). Since MTF moves $x_i$ to the front in $L_i$, it can only introduce additional flipped pairs between $x_i$ and items that are in front of $x_i$ in $L_{i-1}^\star$. The number of these items is exactly $s_i - |X_i|$ and must, by definition, be at most $s_i^\star$, hence we get

$$s_i - |X_i| \leq s_i^\star. \tag{4.2}$$

On the other hand, all flipped pairs between $x_i$ and items in $X_i$ get repaired in $L_i$ by moving $x_i$ to the front. Overall, the change in potential caused by MTF is $\Delta\Phi_i^{\text{MTF}}(\sigma) = s_i - 2\,|X_i|$.


Figure 6: Schematic illustration of the potential change during MTF's move in step $i$. The items are marked according to whether they contribute positively, negatively, or not at all to $\Delta\Phi_i^{\text{MTF}}(\sigma)$ when $x_i$ is moved to the front.

Now consider OPT's action in step $i$, after MTF already moved $x_i$ to the front of its list. If OPT moves $x_i$ forward, this is guaranteed to decrease the number of flipped pairs, as $x_i$ is at the very front in MTF's list. Every paid action, however, can increase the number of flipped pairs by at most one. The change in potential due to OPT's decisions is therefore bounded by $\Delta\Phi_i^{\text{OPT}}(\sigma) \leq p_i^\star$. Overall, we obtain

$$\Delta\Phi_i(\sigma) = \Delta\Phi_i^{\text{MTF}}(\sigma) + \Phi_i^{\text{OPT}}(\sigma) \leq s_i - 2\,|X_i| + p_i^\star.$$

For the amortized cost this implies

$$\begin{aligned} a_i(\sigma) &= \text{MTF}_i(\sigma) + \Delta\Phi_i(\sigma) \\ &\leq s_i + s_i - 2\,|X_i| + p_i^\star \\ &\overset{(4.2)}{\leq} 2s_i^\star + p_i^\star \\ &\leq 2 \cdot \text{OPT}_i(\sigma). \end{aligned} \tag{4.3}$$

We can immediately apply Lemma 4.2 with $\alpha = \beta = 0$ to obtain that MTF is strictly 2-competitive, i.e., MTF does not require more than twice the number of operations as the best offline solution that knows the order in which items will be requested ahead of time. □

## 4.3 Lower bound via averaging

Now that we have an upper bound on the competitive ratio of MTF, it is natural to ask whether there are better algorithms than using the simple MTF heuristic. In other words, we are looking for a lower bound on the competitive ratio $\text{ALG}(\sigma)/\text{OPT}(\sigma)$ of the list update problem. This means that, for every possible online algorithm ALG, we need an instance $\sigma \in \Sigma$ together with a lower bound on $\text{ALG}(\sigma)$ and an upper bound on $\text{OPT}(\sigma)$. We will first derive a general upper bound on the cost of the offline optimum and then construct a bad instance for any given online algorithm.

In Section (2.2) we obtained an upper bound by explicitly analyzing the behavior of OPT. Without understanding the structure of optimum offline solutions or even optimum offline algorithms, it is generally difficult to derive good bounds on $\text{OPT}(\sigma)$ that depend only on $n$ but are otherwise independent of the request sequence $\sigma$. We use an averaging argument – a powerful technique that is often used to obtain upper bounds on the cost of optimum solutions. The general idea of averaging is the following:

1. Introduce a uniform class $\mathcal{A}$ of offline algorithms with a reasonably simple structure.
2. For any given instance $\sigma$, compute the average cost $c_{\text{avg}}$ over all algorithms in $\mathcal{A}$. This average should ideally be independent of $\sigma$, since we chose a uniform class of algorithms, and it should be easy to compute, since we chose a class of simple structure.
3. Use the fact that there is at least one algorithm $\text{ALG} \in \mathcal{A}$ with $\text{ALG}(\sigma) \leq c_{\text{avg}}$ and that $\text{OPT}(\sigma) \leq \text{ALG}(\sigma)$ to get the bound $\text{OPT}(\sigma) \leq c_{\text{avg}}$.

To illustrate, we apply this technique to the list update problem. We consider the simple and uniform class of all algorithms that order the list into one of the $l!$ possibilities using paid actions in the first step, and then never change the order again. By definition, there are $l!$ such algorithms. The maximum cost in step 1 over all these algorithms is $\frac{l(l-1)}{2}$, hence the average setup cost $\alpha(l) < \frac{l(l-1)}{2}$ only depends on the number $l$ of elements in the list.

As the second step of the averaging technique, we compute the average cost $c_{\text{avg}}$ over all $l!$ algorithms for any input $\sigma = (x_1, \ldots, x_n)$. In every step $i$, there are exactly $(l-1)!$ algorithms that have $x_i$ at position $j$, for every $j \in \{1, \ldots, l\}$. The average cost $c_{\text{avg},i}$ of step $i \geq 2$ over all algorithms in our class therefore is

$$c_{\text{avg},i} = \frac{1}{l!} \sum_{j=1}^{l} (j-1)(l-1)! = \frac{(l-1)!}{l!} \cdot \frac{l(l-1)}{2} = \frac{l-1}{2}.$$

Note that we chose our class of algorithms in such a way that $c_{\text{avg},i}$ is independent of both $i$ and $\sigma$ and is easy to compute. This is the secret behind the power of the averaging technique!

To bound the cost of OPT it now only remains to compute $c_{\text{avg}}$, since at least one algorithm of our class does not exceed the average cost and, hence, neither does OPT. We get

$$\text{OPT}(\sigma) \leq c_{\text{avg}} = \alpha(l) + \sum_{i=2}^{n} c_{\text{avg},i} = (n-1) \cdot \frac{l-1}{2} + \alpha(l) = n \cdot \frac{l-1}{2} + \alpha'(l). \tag{4.4}$$

Now that we have a general upper bound on $\text{OPT}(\sigma)$, we need to ask ourselves how the adversary can sabotage any given online algorithm ALG. An obvious request sequence that comes to mind is the sequence that always requests the last element on ALG's list, which forces a total cost of $\text{ALG}(\sigma) = n(l-1) \to_{n\to\infty} \infty$. Together with eq. (4.4) this yields

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} \geq \frac{n(l-1)}{n(l-1)/2 + \alpha'(l)} = \frac{2(l-1)}{(l-1) + \alpha'(l)/n} \underset{n\to\infty}{\to} 2.$$

By Proposition 1.12, we can get arbitrarily close to 2 with our lower bound on the competitive ratio of ALG. Since ALG was chosen arbitrarily, we obtain the following.

**Theorem 4.5** (Karp and Raghavan [1990])**.** *Every deterministic algorithm for the list update problem has competitive ratio at least* 2*.*

**Corollary 4.6.** *The list update problem has competitive ratio* 2*.*

*Proof.* This follows directly from Theorems 4.4 and 4.5. □

## 4.4 Upper bounds via phase partitioning

In Section 4.2 we employed amortized analysis in order to avoid understanding the evolution of MTF's list during the course of the algorithm. In this section, we use phase partitioning (cf. Section 2.4) to explicitly analyze this evolution. Importantly, we will first break the general case down to understanding lists of only two elements.

To this end, we let the cost $\text{ALG}_{xy}(\sigma)$ of an algorithm ALG *induced by* $(x, y) \in X^2$ be defined as the number of steps in which $y$ is accessed and $x$ precedes $y$ in ALG's list, i.e., $\text{ALG}_{xy}(\sigma)$ is the part of the cost for accessing $y$ that is "caused by" $x$. Observe that, by definition,

$$\text{ALG}(\sigma) = \sum_{(x,y)\in X^2} \text{ALG}_{xy}(\sigma) + \sum_{i=1}^{n} p_i. \tag{4.5}$$

In the same vein, for any sequence of items, we obtain the *projection* onto $\{x, y\} \subseteq X$ by removing all occurrences of items in $X \setminus \{x, y\}$. In this way, let $\sigma_{xy}$ denote the projection of $\sigma$ onto $\{x, y\} \subseteq X$, let $L_{xy}$ denote the projection of $L$ onto $\{x, y\} \subseteq X$, etc. Finally, for convenience, we abuse notation and write $\text{ALG}[\sigma_{xy}]$, $\text{ALG}(\sigma_{xy})$ to denote the solution produced by ALG and its cost, respectively, when executed on list $L_{xy}$ with input $\sigma_{xy}$.

**Definition 4.7.** A (deterministic or randomized) list update algorithm ALG satisfies the *pairwise property* if, for all $\sigma \in \Sigma$ and $x, y \in X$, $\text{ALG}_{xy}(\sigma) + \text{ALG}_{yx}(\sigma) = \text{ALG}(\sigma_{xy})$.

Intuitively, the pairwise property states that the costs induced by any pair of elements are fully captured already in the projected instance. Observe that this need not be the case in general (exercise). We give an alternative characterization of the pairwise property.

**Lemma 4.8.** *If an online algorithm* ALG *does not use paid exchanges, it satisfies the pairwise property if and only if, for every* $\sigma \in \Sigma$*, every* $x, y \in X$ *and every* $i \in \{1, \ldots, |\sigma_{xy}| - 1\}$*, the relative order of* $x$ *and* $y$ *between the* $i$*-th access to* $\{x, y\}$ *and the next is the same in* $\text{ALG}[\sigma]$ *and* $\text{ALG}[\sigma_{xy}]$*.*

*Proof.* Since ALG does not use paid exchanges, the relative order of $x$ and $y$ can only change in steps $i$ where $x_i \in \{x, y\}$ is accessed. It is obvious that if $x$ and $y$ are in different relative orders in $\text{ALG}[\sigma]$ and $\text{ALG}[\sigma_{xy}]$ at the beginning of step $i$, then $\text{ALG}_{xy,i}(\sigma) + \text{ALG}_{yx,i}(\sigma) \neq \text{ALG}_i(\sigma_{xy})$. Let $x_i$ be the first request to $\{x, y\}$ where this occurs. Then, since ALG does not use paid exchanges, we have

$$\begin{aligned}
\text{ALG}_{xy}(\sigma_{\le i}) + \text{ALG}_{yx}(\sigma_{\le i}) &= \text{ALG}_{xy}(\sigma_{\le i-1}) + \text{ALG}_{yx}(\sigma_{\le i-1}) + \text{ALG}_{xy,i}(\sigma_{\le i}) + \text{ALG}_{yx,i}(\sigma_{\le i}) \\
&= \text{ALG}((\sigma_{\le i-1})_{xy}) + \text{ALG}_{xy,i}(\sigma_{\le i}) + \text{ALG}_{yx,i}(\sigma_{\le i}) \\
&\neq \text{ALG}((\sigma_{\le i-1})_{xy}) + \text{ALG}_i((\sigma_{\le i})_{xy}) \\
&= \text{ALG}((\sigma_{\le i})_{xy}),
\end{aligned}$$

and the pairwise property is violated for $\sigma_{\le i}$.

On the other hand, assume that ALG violates the pairwise property and let $\sigma \in \Sigma$ be a request sequence of minimum length $n$ for which this is the case. By definition, $\text{ALG}_{x_n y,n}(\sigma) + \text{ALG}_{yx_n,n}(\sigma) \neq \text{ALG}_n(\sigma_{x_n y})$ for some $y \in X$. But this can only be the case if $x_n$ and $y$ are in different relative orders at the beginning of step $n$ in $\text{ALG}[\sigma]$ and $\text{ALG}[\sigma_{xy}]$, since ALG does not use paid exchanges. □

**Corollary 4.9.** MTF *satisfies the pairwise property.*

*Proof.* This follows immediately by Lemma 4.8, since the relative order of two items $x, y \in X$ in MTF's list only depends on which of the two elements was accessed most recently (and on $L_0$). □

We are now ready to formulate a condition under which it is sufficient to analyze the behavior of an algorithm on lists of length two. Using this approach is often referred to as the *list factoring technique.*

**Theorem 4.10** (Bentley and McGeoch [1985])**.** *Let* ALG *be a (deterministic or randomized) online algorithm that does not use paid exchanges and that satisfies the pairwise property, and let* $\rho \geq 1$*. Then,* ALG *is (strictly)* $\rho$*-competitive if and only if it is (strictly)* $\rho$*-competitive on lists of length two.*

*Proof.* For the non-trivial part of the theorem, we assume that ALG is $\rho$-competitive on lists of length two for some $\rho \geq 1$, and show that this implies $\rho$-competitiveness in general. In particular, for a deterministic algorithm ALG, assume there exists $\alpha \in \mathbb{R}$ such that we have

$$\text{ALG}(\sigma_{xy}) \leq \rho \cdot \text{OPT}(\sigma_{xy}) + \alpha, \tag{4.6}$$

for all $x, y \in X$.

Fix two items $x, y \in X$ and consider the offline solution $\text{OFF}[\sigma_{xy}]$ on the projected instance $\sigma_{xy}, L_{xy}$ that behaves as follows. If $\text{OPT}[\sigma]$ switches the relative order of $x$ and $y$ by using a free action during the $k$-th access to an item in $\{x, y\}$, then $\text{OFF}[\sigma_{xy}]$ does the same on the projected instance. If $\text{OPT}[\sigma]$ uses a paid exchange to switch $x$ and $y$ between two accesses to items in $\{x, y\}$, then $\text{OFF}[\sigma_{xy}]$ does the same between the corresponding steps in $\sigma_{xy}$. By definition, $\text{OFF}[\sigma_{xy}]$ maintains the same relative order of $x$ and $y$ and uses the same number of paid exchanges between $x$ and $y$ as $\text{OPT}[\sigma]$. Hence,

$$\text{OPT}(\sigma_{xy}) \leq \text{OFF}(\sigma_{xy}) \leq \text{OPT}_{xy}(\sigma) + \text{OPT}_{yx}(\sigma) + p^\star_{xy}, \tag{4.7}$$

where $p^\star_{xy}$ denotes the number of paid exchanges between $x$ and $y$ in $\text{OPT}[\sigma]$.

Now, since ALG does not use paid exchanges (i.e., $p_i = 0$) and satisfies the pairwise property, we obtain

$$\begin{aligned}
\text{ALG}(\sigma) &\overset{(4.5)}{=} \sum_{(x,y)\in X^2} \text{ALG}_{xy}(\sigma) \\
&= \sum_{\{x,y\}\subseteq X} (\text{ALG}_{xy}(\sigma) + \text{ALG}_{yx}(\sigma)) \\
&\overset{\text{Def. 4.7}}{=} \sum_{\{x,y\}\subseteq X} \text{ALG}(\sigma_{xy}) \\
&\overset{(4.6)}{\leq} \sum_{\{x,y\}\subseteq X} (\rho \cdot \text{OPT}(\sigma_{xy}) + \alpha) \\
&\overset{(4.7)}{\leq} \rho \sum_{\{x,y\}\subseteq X} (\text{OPT}_{xy}(\sigma) + \text{OPT}_{yx}(\sigma) + p^\star_{xy}) + \frac{l(l-1)}{2}\alpha \\
&\overset{(4.5)}{=} \rho \cdot \text{OPT}(\sigma) + \alpha'(l),
\end{aligned}$$

with $\alpha'(l) \in \mathbb{R}$. Hence ALG is $\rho$-competitive on $\sigma$, as claimed. If ALG is strictly $\rho$-competitive on instances of length two, then $\alpha = 0$ and $\alpha'(l) = 0$, thus ALG is strictly $\rho$-competitive on $\sigma$.

For randomized algorithms, the proof is identical if we replace all costs by expected costs. □

This provides an alternative way to upper bound the competitive ratio of MTF.

**Corollary 4.11.** MTF *is strictly* 2*-competitive.*

*Proof.* By Theorem 4.10, it suffices to show that MTF is 2-competitive on lists of length two. To see this, consider a request sequence $\sigma = (x_1, \ldots, x_n)$ for the list update problem with $|X| = 2$ items. Let $i_1 < \cdots < i_m$ with $m \leq n$ denote the steps in which MTF has non-zero cost, i.e., $\text{MTF}_{i_j}(\sigma) = 1$ for $j \in \{1, \ldots, m\}$, and $\text{MTF}(\sigma) = m$. For convenience, let $i_0 := 0$. Observe that $x_{i_j} \neq x_{i_{j+1}}$ for $j \in \{1, \ldots, m-1\}$ and $x_{i_1}$ must be in second position in $L_0$. It follows that $\sum_{i=i_j}^{i_{j+1}} \text{OPT}_i(\sigma) \geq 1$ for $j \in \{0, \ldots, m\}$. With this, we have

$$\begin{aligned}
\text{OPT}(\sigma) &= \sum_{i=1}^{n} \text{OPT}_i(\sigma) \\
&\geq \sum_{j=0}^{\lfloor (m-1)/2 \rfloor} \sum_{i=i_{2j}}^{i_{2j+1}} \text{OPT}_i(\sigma) \\
&\geq \lfloor (m-1)/2 \rfloor + 1 \\
&\geq m/2 \\
&= \frac{1}{2}\text{MTF}(\sigma).
\end{aligned}$$

□

### 4.4.1 The timestamp algorithm

The list factoring technique allows to analyze algorithms that are more involved than MTF. For example, consider the algorithm TIMESTAMP that is defined as follows. If an item $x \in X$ is accessed for the first time, TIMESTAMP does not change the order of the list. Each subsequent time $x \in X$ is accessed, TIMESTAMP moves $x$ in front of the first item on its list that was accessed at most once since the previous access to $x$. We will see that TIMESTAMP is a competitive algorithm. To this end, we employ the list factoring technique.

**Lemma 4.12.** TIMESTAMP *satisfies the pairwise property.*

*Proof.* We use the characterization of the pairwise property of Lemma 4.8. For the sake of contradiction, assume that there is a request sequence $\sigma \in \Sigma$ and two items $\{x, y\} \in X$ that do not remain in the same relative order throughout the execution of TIMESTAMP on $\sigma$ and $\sigma_{xy}$, respectively. Note that, by definition, whenever TIMESTAMP$[\sigma_{xy}]$ changes the order of $x$ and $y$ during an access to $x$ or $y$, so must TIMESTAMP$[\sigma]$ during the corresponding access to the same element. This means that there must be a step $i$ in $\sigma$ with $x_i \in \{x, y\}$, where TIMESTAMP$[\sigma]$ moves $x_i$ in front of $y_i := \{x, y\} \setminus x_i$, but, in the corresponding step in $\sigma_{xy}$, TIMESTAMP$[\sigma_{xy}]$ leaves $y_i$ in front of $x_i$.

First observe that $x_i$ must have been accessed before step $i$, otherwise TIMESTAMP$[\sigma]$ would not move it forward. Let $a$ denote the first item that ends up behind $x_i$ after step $i$ in TIMESTAMP$[\sigma]$. Since TIMESTAMP$[\sigma]$ moves $x_i$ to the front of $y_i$ by assumption, $a$ cannot be behind $y_i$ in the list. If $a = y_i$, then $y_i$ must have been accessed at most once since the previous access to $x_i$, and TIMESTAMP$[\sigma_{xy}]$ would move $x_i$ forward as well, contradicting the choice of $i$. Thus $a \neq y_i$ is in front of $y_i$ in the list of TIMESTAMP$[\sigma]$.

Now let $j' < j < i$ denote the two last steps before step $i$ where $y_i$ was requested, and let $i' < i$ denote the last step where $x_i$ was requested. Since TIMESTAMP$[\sigma_{xy}]$ does not move $x_i$ forward in step $i$, we must have $i' < j'$, i.e., $y_i$ was accessed at least twice since step $i'$. On the other hand, TIMESTAMP$[\sigma]$ moving $x_i$ in front of $a$ in step $i$ implies that $a$ was requested at most once during steps $i' + 1, \ldots, i - 1$, and therefore at most once during steps $j' + 1, \ldots, j - 1$. This means that TIMESTAMP$[\sigma]$ must have moved $y_i$ in front of $a$ in step $j$. Since $a$ is requested at most once during steps $j + 1, \ldots, i - 1$ however, TIMESTAMP$[\sigma]$ cannot haved moved it back in front of $y_i$ afterwards. This is a contadiction with $a$ being in front of $y_i$ in the list of TIMESTAMP$[\sigma]$ in step $i$. □

To analyze the behavior of an algorithm on lists of length two, the following phase partitioning will be useful. Note that we can extend any request sequence so that it starts and ends with two consecutive accesses to the same element, without affecting $\textsc{Opt}(\sigma)$.

**Definition 4.13.** The (unique) *alternating phase partitioning* of a request sequence $\sigma_{xy} = (x_1, \ldots, x_n) \in \{x, y\}^n$ for $L_0 = (x, y)$ is given by $\sigma'_{xy} = \pi_0 \oplus \cdots \oplus \pi_N = (x, x) \oplus \sigma_{xy} \oplus (z)^k$, where $z$ denotes the element in front of the list at the end of $\textsc{Opt}[\sigma_{xy}]$, $k \in \{1, 2\}$ is defined such that the number of consecutive requests to $z$ at the end of $\sigma'_{xy}$ is even, and each phase $\pi_j$ for $j \in \{0, \ldots, N\}$ is defined such that $\pi_j = (a_1, a_2, \ldots, a_m)$ with $a_1 \neq a_2 \neq \cdots \neq a_{m-1} = a_m$. Note that $\textsc{Opt}(\sigma'_{xy}) = \textsc{Opt}(\sigma_{xy})$.

For example, the alternating phase partitioning of $\sigma_{xy} = (x, y, x, x, x, y, x, y, x, x, y, y, y)$ for $L_0 = (x, y)$ is $\pi_0 \oplus \cdots \oplus \pi_4 = (x, x) \oplus \sigma_{xy} \oplus (y)$ with $\pi_0 = (x, x)$, $\pi_1 = (x, y, x, x)$, $\pi_2 = (x, y, x, y, x, x)$, $\pi_3 = \pi_4 = (y, y)$.

We now use the alternating phase partitioning to analyze Timestamp.

**Theorem 4.14** (Albers [1998])**.** Timestamp *is strictly* 2*-competitive.*

*Proof.* By Theorem 4.10 and Lemma 4.12, it is sufficient to show that Timestamp is strictly 2-competitive on lists of length two. To this end, let $\sigma_{xy} \in \Sigma$ be a request sequence for a list with $X = \{x, y\}$ and consider the alternating phase partitioning $\sigma'_{xy} = \pi_0 \oplus \cdots \oplus \pi_N$ of $\sigma_{xy}$. Observe that $\textsc{Timestamp}(\sigma_{xy}) \leq \textsc{Timestamp}(\sigma'_{xy})$ and $\textsc{Opt}(\sigma'_{xy}) = \textsc{Opt}(\sigma_{xy})$, which means that it suffices to show $\textsc{Timestamp}(\sigma'_{xy}) \leq 2\,\textsc{Opt}(\sigma'_{xy})$. We fix Opt to be the algorithm that changes the order of the items exactly when the second item in the list is accessed and the next request is for the same item (note that we can adapt any solution to be of this form without increasing its cost). We abuse notation and let $\textsc{Timestamp}(\pi_j)$ and $\textsc{Opt}(\pi_j)$ denote the costs of Timestamp and Opt, respectively, incurred during phase $\pi_j$ for $j \in \{0, \ldots, N\}$. We claim that $\textsc{Timestamp}(\pi_j) \leq 2 \cdot \textsc{Opt}(\pi_j)$ for every phase $\pi_j$. This completes the proof, since then

$$\textsc{Timestamp}(\sigma'_{xy}) = \sum_{j=0}^{N} \textsc{Timestamp}(\pi_j) \leq 2 \sum_{j=0}^{N} \textsc{Opt}(\pi_j) = 2\,\textsc{Opt}(\sigma'_{xy}).$$

To prove our claim for $\pi_0$, observe that $\textsc{Opt}(\pi_0) = \textsc{Timestamp}(\pi_0) = 0$.

Now consider a phase $\pi_j$ with $j \in \{1, \ldots, N\}$ and assume, without loss of generality, that $\pi_{j-1}$ ends with a request to $x$. Note that by definition of both Timestamp and Opt, the item $x$ is the first item in both Opt' and Timestamp's list at the start of phase $\pi_j$. There are four cases, depending on whether $\pi_j$ starts with $x$ or $y$ and whether it ends with $x$ or $y$. The following table lists the costs incurred by Timestamp and Opt in each case, where $k \in \mathbb{N} \cup \{0\}$. Observe that in each case $\textsc{Timestamp}(\pi_j) \leq 2 \cdot \textsc{Opt}(\pi_j)$, as claimed.

| form of phase $\pi_j$ | $\textsc{Timestamp}(\pi_j)$ $(k > 0)$ | $\textsc{Timestamp}(\pi_j)$ $(k = 0)$ | $\textsc{Opt}(\pi_j)$ |
|---|---|---|---|
| $x(yx)^k x$ | $2k - 1$ | 0 | $k$ |
| $x(yx)^k yy$ | $2k$ | 1 | $k + 1$ |
| $yx(yx)^k x$ | $2k + 1$ | 1 | $k + 1$ |
| $(yx)^k yy$ | $2k$ | 1 | $k + 1$ |

□

## 4.5 Randomized list update algorithms

In Theorem 4.5, we have seen that no deterministic list update algorithm can be better than 2-competitive. The proof used a request sequence that depends on the behavior of the online algorithm by repeatedly accessing the item currently at the end of the list. Similarly to the paging problem (cf. Section 3.1), we may try to use randomization in order to be less predictable and

improve on MTF's performance on instances of this kind. Recall that we assumed the adversary to be oblivious, i.e., it does not know the random choices of our algorithm when constructing its worst-case instance.

A natural way to randomize MTF would be to move accessed items to the front independently with probability 1/2. Note that this algorithm RMTF does not formally fall under Definition 3.1 of a randomized algorithm, since it cannot be expressed as a superposition of deterministic algorithms. In addition, RMTF does not improve on the performance of MTF at all.

**Proposition 4.15.** *The competitive ratio*[3] *of* RMTF *is at least* 2.

*Proof.* Let $L_0 = (x, y)$ and consider the adversarial request sequence $\sigma_{\leq n}^{(y)} = (y, y, \dots, y)$ of length $n \in \mathbb{N}$. The offline optimum solution for this instance is $\text{MTF}[\sigma_{\leq n}^{(y)}]$ and has cost

$$\text{OPT}(\sigma_{\leq n}^{(y)}) = \text{MTF}(\sigma_{\leq n}^{(y)}) = |X| - 1.$$

Let $I$ be the indicator variable for the step in which RMTF moves $y$ to the front. Then

$$\mathbb{E}[\text{RMTF}(\sigma_{\leq n}^{(y)})] = \sum_{i=1}^{n} \Pr[I \geq i] = \sum_{i=1}^{n} \frac{1}{2^{i-1}} = \frac{1 - 1/2^n}{1 - 1/2} = 2 - \frac{1}{2^{n-1}} \to_{n\to\infty} 2 \cdot \text{OPT}(\sigma_{\leq n}^{(y)}).$$

In order to apply Proposition 3.5, we need to ensure that $\lim_{n\to\infty} \text{OPT}(\sigma) = \infty$. We can achieve that similarly to the proof of Proposition (3.4), by using the request sequence $\sigma_{\leq n}^{(y)} \oplus \sigma_{\leq n}^{(x)} \oplus \sigma_{\leq n}^{(y)} \oplus \dots$, where $\sigma_{\leq n}^{(x)} = (x, x, x, \dots) \in X^n$. □

If we aim at defining a proper randomized algorithm (according to Definition 3.1) inspired by MTF, we have to express it as a superposition of deterministic algorithms. This means that all random choices must be made right at the start of the algorithm. We introduce an algorithm BIT that achieves this by initially setting a bit $b(x) \in \{0, 1\}$ randomly and with equal probability for every item $x \in X$. Whenever an item $x \in X$ is accessed, BIT flips the value of $b(x)$ and then moves $x$ to the front of the list if $b(x) = 1$. We show that BIT indeed improves the competitive ratio of MTF.

**Theorem 4.16** (Reingold et al. [1994])**.** BIT *is strictly* 1.75*-competitive.*

*Proof.* Observe that, during execution of BIT for a request sequence $\sigma$, the relative order of two elements only depends on $\sigma_{xy}$. In other words, BIT satisfies the pairwise property (by Lemma 4.8). Since BIT does not use paid exchanges, we can therefore use the list factoring technique (Theorem 4.10) to restrict our analysis to request sequences $\sigma_{xy}$ for lists of size two. As in the proof of Theorem 4.14, we fix OPT to be the algorithm that changes the order of the items exactly when the second item is accessed and the next request is for the same item.

To this end, let $\sigma'_{xy} = \pi_0 \oplus \dots \oplus \pi_N$ be the alternating phase partitioning of $\sigma_{xy}$. Note that $\text{BIT}(\sigma'_{xy}) \geq \text{BIT}(\sigma_{xy})$ and $\text{OPT}(\sigma'_{xy}) = \text{OPT}(\sigma_{xy})$, which means that it suffices to show $\text{BIT}(\sigma'_{xy}) \leq 1.75 \cdot \text{OPT}(\sigma'_{xy})$. As in the proof of Theorem 4.14, it suffices to show that $\mathbb{E}[\text{BIT}(\pi_j)] \leq \frac{7}{4} \cdot \text{OPT}(\pi_j)$ for every phase $\pi_j$. By definition of the alternating phase partitioning, the only item accessed during phase $\pi_0$ is already at the front of $L_0$, thus $\text{OPT}(\pi_0) = \text{BIT}(\pi_0) = 0$.

Now, consider any phase $\pi_j$ with $j \in \{1, \dots, N\}$. Let $x \in X$ be the item last requested in phase $\pi_{j-1}$ and $y \in X$ be the other item. Observe that at the start of phase $\pi_j$, the item $x$ is in front in both OPT's and BIT's list, and the values of $b(x)$ and $b(y)$ are both independently 0 or 1 with probability 1/2. Let $\mathbb{E}[\text{BIT}_{xx}(\sigma')]$ denote the expected cost of BIT for the subsequence $\sigma'$, assuming

[3] Here we define the competitive ratio analogously to Def 3.2, where the expectation $\mathbb{E}[\text{RMTF}(\sigma)]$ is taken over the random choices of RMTF.

that that $x$ was requested in the two previous steps. Then, using the abbreviation $ab \equiv (a, b)$, we have

$$\mathbb{E}[\text{BIT}_{xx}(xy)] = 1,$$
$$\mathbb{E}[\text{BIT}_{xx}(yx)] = \mathbb{E}[\text{BIT}_{xx}(yy)] = 1 + \frac{1}{2} = \frac{3}{2}.$$

Let $\mathbb{E}[\text{BIT}_{xyx}(\sigma')]$ denote the expected cost of BIT for the request sequence $\sigma'$, assuming that the previously requested items were $x$, $y$, $x$ in this order, and let $\mathbb{E}[\text{BIT}_{yxy}(\sigma')]$ be defined analogously. Observe that the probability that $y$ is in front of BIT's list after the subsequence $(x, y, x)$ is $1/4$, independently of the earlier requests, and symmetrically for $x$ and $(y, x, y)$. Hence,

$$\mathbb{E}[\text{BIT}_{xyx}(y)] = \mathbb{E}[\text{BIT}_{yxy}(x)] = \frac{3}{4},$$
$$\mathbb{E}[\text{BIT}_{xyx}(x)] = \mathbb{E}[\text{BIT}_{yxy}(y)] = \frac{1}{4},$$
$$\mathbb{E}[\text{BIT}_{xyx}(yy)] = \mathbb{E}[\text{BIT}_{xyx}(y)] + \mathbb{E}[\text{BIT}_{yxy}(y)] = \frac{3}{4} + \frac{1}{4} = 1,$$
$$\mathbb{E}[\text{BIT}_{xyx}((yx)^k)] = 2k \cdot \frac{3}{4} = \frac{3}{2}k,$$
$$\mathbb{E}[\text{BIT}_{xx}((yx)^k)] = \frac{3}{2} + \frac{3}{2}(k-1) = \frac{3}{2}k.$$

With this, we have the following costs, depending on the first and last request of $\pi_j$, assuming that the previous phase ended with two requests for $x \in X$. In particular, we have $\mathbb{E}[\text{BIT}(\pi_j)] \leq \frac{7}{4} \cdot \text{OPT}(\pi_j)$ in each case.

| form of phase $\pi_j$ | $\mathbb{E}[\text{BIT}(\pi_j)]$ $(k > 0)$ | $\mathbb{E}[\text{BIT}(\pi_j)]$ $(k = 0)$ | $\text{OPT}(\pi_j)$ |
|---|---|---|---|
| $x(yx)^k x$ | $0 + \frac{3}{2}k + \frac{1}{4} = \frac{1}{4} + \frac{3}{2}k$ | $0$ | $k$ |
| $x(yx)^k yy$ | $0 + \frac{3}{2}k + 1 = 1 + \frac{3}{2}k$ | $1 + \frac{1}{2} = \frac{3}{2}$ | $k+1$ |
| $yx(yx)^k x$ | $\frac{3}{2} + \frac{3}{2}k + \frac{1}{4} = \frac{7}{4} + \frac{3}{2}k$ | $\frac{3}{2} + \frac{1}{4} = \frac{7}{4}$ | $k+1$ |
| $(yx)^k yy$ | $\frac{3}{2} + \frac{3}{2}(k-1) + 1 = 1 + \frac{3}{2}k$ | $\frac{3}{2}$ | $k+1$ |

□

If we compare the performance of BIT to the performance of TIMESTAMP for each possible phase in terms of competitive ratios, we observe that both algorithms have their worst-cases in different types of phases:

| | $\mathbb{E}[\text{BIT}(\pi_j)]/\text{OPT}(\pi_j)$ | | | | $\text{TIMESTAMP}(\pi_j)/\text{OPT}(\pi_j)$ | | | |
|---|---|---|---|---|---|---|---|---|
| form of phase $\pi_j$ | $k=0$ | $k=1$ | $k=2$ | $k \geq 2$ | $k=0$ | $k=1$ | $k=2$ | $k \geq 2$ |
| $x(yx)^k x$ | – | **1.75** | 1.63 | $\in [1.5, 1.63]$ | – | 1 | 1.5 | $\in [1.5, \mathbf{2}]$ |
| $x(yx)^k yy$ | 1.5 | 1.25 | 1.33 | $\in [1.33, 1.5]$ | 1 | 1 | 1.33 | $\in [1.33, \mathbf{2}]$ |
| $yx(yx)^k x$ | **1.75** | 1.63 | 1.58 | $\in [1.5, 1.58]$ | 1 | 1.50 | 1.67 | $\in [1.67, \mathbf{2}]$ |
| $(yx)^k yy$ | 1.5 | 1.25 | 1.33 | $\in [1.33, 1.5]$ | 1 | 1 | 1.25 | $\in [1.25, \mathbf{2}]$ |

Linearity of expectation allows us to interpolate linearly between the performance of BIT and TIMESTAMP. We define the randomized algorithm COMB as a superposition of BIT and TIMESTAMP. More precisely, COMB initially selects BIT with probability 4/5 and TIMESTAMP with probability 1/5.

**Theorem 4.17** (Albers et al. [1995])**.** COMB *is strictly* 1.6*-competitive.*

*Proof.* By Lemma 4.12 and since BIT satisfies the pairwise property, any superposition of BIT and TIMESTAMP, including COMB, has the pairwise property. This means that, as before, we can

use list factoring (Theorem 4.10) and restrict our analysis to phases $\pi_j$ of the alternating phase partitioning of request sequences on lists of length two. Because of

$$\mathbb{E}[\textsc{Comb}(\pi_j)] = \mathbb{E}[\frac{4}{5}\mathbb{E}[\textsc{Bit}(\pi_j)] + \frac{1}{5}\textsc{Timestamp}(\pi_j)] = \frac{4}{5}\mathbb{E}[\textsc{Bit}(\pi_j)] + \frac{1}{5}\textsc{Timestamp}(\pi_j),$$

we obtain the values $\mathbb{E}[\textsc{Comb}(\pi_j)]/\textsc{Opt}(\pi_j)$ as a linear combination of the corresponding values for Bit and Timestamp. For the first phase $\pi_0$, we have

$$\mathbb{E}[\textsc{Comb}(\pi_0)] = (\frac{4}{5} \cdot \mathbb{E}[\textsc{Bit}(\pi_0)] + \frac{1}{5} \cdot \textsc{Timestamp}(\pi_0)) = 0 = \textsc{Opt}(\pi_0).$$

Regarding the other phases, the following table contains all linear combinations of the above tables for Bit and Timestamp (assuming that the previous phase ended with two requests for $x \in X$). Observe that the competitive ratio is indeed bounded by 1.6, and that this bound is tight in many cases.

| | $\mathbb{E}[\textsc{Comb}(\pi_j)]/\textsc{Opt}(\pi_j)$ | | | |
|---|---|---|---|---|
| form of phase $\pi_j$ | $k=0$ | $k=1$ | $k=2$ | $k \geq 2$ |
| $x(yx)^k x$ | – | **1.6** | **1.6** | **1.6** |
| $x(yx)^k yy$ | 1.4 | 1.2 | 1.33 | $\in [1.33, \mathbf{1.6}]$ |
| $yx(yx)^k x$ | **1.6** | **1.6** | **1.6** | **1.6** |
| $(yx)^k yy$ | 1.4 | 1.2 | 1.31 | $\in [1.31, \mathbf{1.6}]$ |

□

## 4.6 Lower bound for randomized list update

Now that we established a good upper bound for randomized list update in the previous section, we derive a lower bound using Yao's principle (cf. Section 3.2).

**Theorem 4.18** (Teia [1993])**.** *Every randomized list update algorithm has competitive ratio at least* 1.5*.*

*Proof.* In order to apply Yao's principle, we define a sequence $(q_N \colon \Sigma \to [0,1])_{N\in\mathbb{N}}$ of probability distributions for a set of items $X$ with $l := |X|$. For each $N \in \mathbb{N}$, we describe $q_N$ by a randomized construction of a request sequence $\sigma = \pi_1 \oplus \cdots \oplus \pi_N \in X^n$ consisting of $N$ phases. The construction of the phases uses a simple offline algorithm Off that moves a requested item to the front of the list if and only if this item is requested again later in the same phase. Every phase $\pi_j$, $j \in \{1, \ldots, N\}$ is defined in the following way.

We let $L'_i$ denote Off's list *at the beginning of* step $i$. Let $i_j$ denote the first step of phase $\pi_j$ and set $i_0 := 0$, $i_{N+1} = n+1$. Observe that, by definition of Off, the order of the list $L'_{i_j}$ at the start of phase $\pi_j$ only depends on the previous phases. This allows us to base the construction of phase $\pi_j$ on $L'_{i_j}$ in the following way. In phase $\pi_j$ we simply request all items one by one in the order given by $L'_{i_j}$. Each item is either requested a single time or three times in a row, and both cases occur with probability $1/2$. By definition of Off, its cost in phase $\pi_j$ is given by exactly

$$\textsc{Off}(\pi_j) := \sum_{i=1}^{l}(i-1) = l(l-1)/2. \tag{4.8}$$

Now fix any deterministic online algorithm $\textsc{Alg} \in \mathcal{A}_{\text{det}}$ and let $L_i$ denote Alg's list *at the beginning of* step $i$. We assume that Alg does not use paid exchanges for now and argue later why we may do this. Consider the potential function $\Phi_i(\sigma) := \sum_{\{x,y\}\subseteq X} \Phi_i^{xy}$ *at the beginning of* step $i$ (analogous to the one of Section 4.2), with

$$\Phi_i^{yx} \equiv \Phi_i^{xy} := \begin{cases} 0 & \text{if } x \text{ and } y \text{ are in the same relative order in } L_i \text{ and } L'_i, \\ 1 & \text{else.} \end{cases}$$

Let $\Delta\Phi(\pi_j) := \Phi_{i_{j+1}}(\sigma) - \Phi_{i_j}(\sigma)$ denote the change in potential during phase $j$, and let $\Delta\Phi(\sigma) = \sum_{j=1}^{N} \Delta\Phi(\pi_j)$ denote the total change in potential. Observe that $\Phi_0(\sigma) = 0$ and $0 \leq \Phi_i(\sigma) \leq l(l-1)/2$ throughout. Therefore,

$$0 \leq \Delta\Phi(\sigma) \leq l(l-1)/2. \tag{4.9}$$

**We claim that $\mathbb{E}_{\sigma\sim q_N}[\Delta\Phi(\pi_j) + \textsc{Alg}(\pi_j)] \geq \frac{3}{4}l(l-1)$.** Application of Yao's principle (Theorem 3.7) then completes the proof, since

$$\begin{aligned}
\limsup_{N\to\infty} \frac{\inf_{\textsc{Alg}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_N}[\textsc{Alg}(\sigma)]}{\mathbb{E}_{\sigma\sim q_N}[\textsc{Opt}(\sigma)]} &\geq \limsup_{N\to\infty} \frac{\inf_{\textsc{Alg}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_N}[\sum_{j=1}^{N} \textsc{Alg}(\pi_j)]}{\mathbb{E}_{\sigma\sim q_N}[\textsc{Off}(\sigma)]} \\
&\geq \limsup_{N\to\infty} \frac{\sum_{j=1}^{N} \frac{3}{4}l(l-1) - \mathbb{E}_{\sigma\sim q_N}[\sum_{j=1}^{N} \Delta\Phi(\pi_j)]}{\sum_{j=1}^{N} \mathbb{E}_{\sigma\sim q_N}[\textsc{Off}(\pi_j)]} \\
&\overset{(4.8)}{=} \limsup_{N\to\infty} \frac{\frac{3}{4}Nl(l-1) - \mathbb{E}_{\sigma\sim q_N}[\Delta\Phi(\sigma)]}{\frac{1}{2}Nl(l-1)} \\
&\overset{(4.9)}{\geq} \frac{3}{2} - \limsup_{N\to\infty} \frac{l(l-1)/2}{\frac{1}{2}Nl(l-1)} \\
&= \frac{3}{2},
\end{aligned}$$

and, by construction, $\lim_{N\to\infty} \mathbb{E}_{\sigma\sim q_N}[\textsc{Opt}(\sigma)] = \infty$.

It remains to show our claim. Let $\textsc{Alg}_{xy}(\pi_j)$ denote the number of requests to $y$ during phase $\pi_j$ in which $x$ is in front of $y$ in Alg's list, i.e., $\textsc{Alg}_{xy}(\pi_j)$ is the cost for Alg of requests to $y$ "caused" by $x$ during phase $\pi_j$. With this, the cost of Alg in phase $j$ can be expressed as

$$\textsc{Alg}(\pi_j) := \sum_{x,y\in X} (\textsc{Alg}_{xy}(\pi_j) + \textsc{Alg}_{yx}(\pi_j)).$$

Fix any phase $\pi_j$ and any two items $x, y \in X$ such that $x$ is in front of $y$ in the list $L_{i_j}$ at the start of $\pi_j$, i.e., in particular, $x$ is requested before $y$ in phase $\pi_j$. Let $\Delta\Phi^{xy}(\pi_j) = \Phi^{xy}_{i_{j+1}} - \Phi^{xy}_{i_j} \in \{-1, 0, 1\}$ denote the change of $\Phi^{xy}$ during phase $\pi_j$. It suffices to show $\mathbb{E}[\Delta\Phi^{xy}(\pi_j) + \textsc{Alg}_{xy}(\pi_j) + \textsc{Alg}_{yx}(\pi_j)] \geq 3/2$, since then

$$\mathbb{E}[\Delta\Phi(\pi_j) + \textsc{Alg}(\pi_j)] = \sum_{x,y\in X} \Big(\mathbb{E}[\Delta\Phi^{xy}(\pi_j) + \textsc{Alg}_{xy}(\pi_j) + \textsc{Alg}_{yx}(\pi_j)]\Big) \geq \frac{3}{4}l(l-1).$$

Let $i_x$, $i_y$ denote the steps in which $x$, respectively $y$, are accessed for the first time during phase $\pi_j$. With this, define $\Delta_x := \Phi^{xy}_{i_y} - \Phi^{xy}_{i_j}$ and $\Delta_y := \Phi^{xy}_{i_{j+1}} - \Phi^{xy}_{i_y}$, so that $\Delta\Phi^{xy}(\pi_j) = \Delta_x + \Delta_y$. Note that, because Alg does not use paid exchanges, $\Phi^{xy}_{i_j} = \Phi^{xy}_{i_x}$. We distinguish two cases.

**Case 1: $x$ is in front of $y$ in $L_{i_y}$,i.e., $\Phi^{xy}_{i_y} = 0$.**
If $\Phi^{xy}_{i_x} = 1$, then $y$ is in front of $x$ in $L_{i_x}$ and $\mathbb{E}[\textsc{Alg}_{yx}(\pi_j)] \geq 1$. This means that, for either value of $\Phi^{xy}_{i_x}$, we have $\Delta_x + \mathbb{E}[\textsc{Alg}_{yx}(\pi_j)] \geq 0$.
Note that the relative order of $x$ and $y$ in $L'_{i_y}$ is still the same as in $L'_{i_j}$, in particular, $x$ is in front of $y$ in both $L_{i_y}$ and $L'_{i_y}$.

**Case 1.1: Alg moves $y$ in front of $x$ in step $i_y$.**
If $y$ is accessed once during phase $\pi_j$, we have $\Delta_y + \textsc{Alg}_{xy}(\pi_j) = 1 + 1 = 2$, otherwise we have $\Delta_y + \textsc{Alg}_{xy}(\pi_j) = 0 + 1 = 1$. Since both cases occur with probability $1/2$, we have $\mathbb{E}[\Delta\Phi^{xy}(\pi_j) + \textsc{Alg}_{xy}(\pi_j) + \textsc{Alg}_{yx}(\pi_j)] \geq \mathbb{E}[\Delta_y + \textsc{Alg}_{xy}(\pi_j)] = 3/2$.

**Case 1.2: Alg leaves $x$ in front of $y$ in step $i_y$.**
If $y$ is accessed once during phase $\pi_j$, we have $\Delta_y + \textsc{Alg}_{xy}(\pi_j) = 0 + 1 = 1$, otherwise we

have $\Delta_y + \text{ALG}_{yx}(\pi_j) \geq -1 + 3 = 2$. Since both cases occur with probability $1/2$, we have $\mathbb{E}[\Delta\Phi^{xy}(\pi_j) + \text{ALG}_{xy}(\pi_j) + \text{ALG}_{yx}(\pi_j)] \geq \mathbb{E}[\Delta_y + \text{ALG}_{xy}(\pi_j)] = 3/2$.

**Case 2: $x$ is behind $y$ in $L_{i_y}$, i.e., $\Phi^{xy}_{i_y} = 1$.**

Since $x$ is in front of $y$ in $L'_{i_y}$ by definition of $\sigma$, $y$ must be in front of $x$ in $L_{i_y}$. Thus $\text{ALG}_{xy}(\pi_j) = 0$. Since ALG does not use paid exchanges, $y$ must already have been ahead of $x$ in $L_{i_x}$. This implies $\Phi^{xy}_{i_x} = 1$ and thus $\Delta_x = 0$.

Because of $\Delta_x = 0$ and because the relative order of $x$ and $y$ in OFF's list does not change until step $i_y$, the same must be true for the order in ALG's list, since moving $x$ in front of $y$ cannot be undone by ALG without paid exchanges before $y$ is accessed. If $x$ is accessed once, we have $\text{ALG}_{yx}(\pi_j) = 1$; if $x$ is accessed three times, we have $\text{ALG}_{yx}(\pi_j) = 3$. Since both cases occur with probability $1/2$, we have $\mathbb{E}[\text{ALG}_{yx}(\pi_j)] = 2$.

Now, if $y$ is accessed once, we have $\Delta_y = 1 - 1 = 0$. If $y$ is accessed three times, OFF moves $y$ ahead of $x$ so that the relative order of $x$ and $y$ becomes the same in both lists. Note again that ALG cannot move $x$ ahead of $y$ without paid exchanges. Thus $\Delta_y = 0 - 1 = -1$ in this case. Since both cases occur with equal probabilities, we obtain $\mathbb{E}[\Delta_y] = -1/2$.

Overall, we have $\mathbb{E}[\Delta\Phi^{xy}(\pi_j) + \text{ALG}_{xy}(\pi_j) + \text{ALG}_{yx}(\pi_j)] = -1/2 + 0 + 2 = 3/2$.

Observe that the above proof estimates the expected cost of ALG by only taking into account the relative positions of pairs of elements. In particular, we do not use in any way that the pairwise orders of the items must be consistent, i.e., a total order. This means that the proof applies even for algorithms that do not need to maintain an ordered list, but only an (arbitrary) relation between pairs of elements. For this kind of algorithms, paid exchanges can be eliminated easily, by deferring each exchange that moves item $x$ ahead of item $y$ to the next time $x$ is requested, without increasing the total cost. Thus our assumption that ALG does not use paid exchanges is without loss of generality. Note that if we require ALG to maintain a total order, this transformation may not be possible, since $x$ and $y$ may no longer be adjacent when $x$ is requested. □

*Remark* 4.19. Note the gap between the bounds of Theorems 4.17 and 4.18. Closing this gap is a prominent open problem in online optimization.

# 5 Metrical Task Systems*

In the previous chapters, we have designed competitive online algorithms that were tailored for specific online problems. Ideally, we would like to generalize these algorithms to be applicable to a wide variety of settings, i.e., to design competitive algorithms for a large class of online problems. Of course, larger classes generally only allow for worse competitive ratios, since we empower the adversary by allowing additional instances. In particular, we have to limit our scope if we want online algorithms with bounded competitive ratios.

Abstractly, all problems we have studied so far have in common that the underlying system (e.g., the cache, the list, ...) is in some *state* after every step, and that the cost in each step only depends on the newest request, the current state (e.g., the position of the requested item on the list), and the state transition in the step (e.g., number of transpositions required to change the order of the list). In contrast, the cost in a step of a request-answer game can depend arbitrarily of previous actions and requests. We define the abstract framework of a *metrical task system* that captures the structure outlined above, with the additional requirement that the costs of state transitions define a metric.

**Definition 5.1.** A *metric space* $\mathcal{M}$ is a pair $(S, d)$, where $S$ is a set of points and $d\colon S \times S \to \mathbb{R}_{\geq 0}$ is a metric distance function that satisfies (for pairwise distinct $s, s', s'' \in S$):

1. $d(s, s) = 0$ (reflexivity)
2. $d(s, s') > 0$ (positivity)
3. $d(s, s') = d(s', s)$ (symmetry)
4. $d(s, s') + d(s', s'') \geq d(s, s'')$ (triangle inequality)

A metrical task system consists of a set of states with (metric) transition costs, as well as a cost associated with responding to a request (or *task*) in a given state.

**Definition 5.2.** A *metrical task system* is a pair $(\mathcal{M}, \mathcal{R})$, where $\mathcal{M} = (S, \tau)$ is a metric space and $\mathcal{R}$ is a set of *tasks* $r\colon S \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. We say that $S$ are *states* and $\tau$ are the *transition costs* between states, and we set $\tau_{\min} := \min_{s \neq s'} \tau(s, s')$ and $\tau_{\max} := \max_{s,s' \in S} \tau(s, s')$.

A metrical task system naturally induces the online problem where tasks arrive online:

| Metrical Task System | |
|---|---|
| **given:** | metrical task system $(\mathcal{M} = (S, \tau), \mathcal{R})$, initial state $s_0 \in S$ |
| **online:** | sequence $\sigma = (r_1, \ldots, r_n) \in \mathcal{R}^n$ of tasks requested in this order |
| **actions** (step $i$)**:** | transition from state $s_{i-1}$ to state $s_i$ (cost: $\tau(s_{i-1}, s_i)$) |
| **objective:** | minimize the total cost $\sum_{i=1}^{n} \textsc{Alg}_i(\sigma) = \sum_{i=1}^{n} [\tau(s_{i-1}, s_i) + r_i(s_i)]$ |

**Example 5.3.** We can model the paging problem by setting $S = \binom{P}{k}$ to be every possible cache configuration, letting $\tau(C, C') = |C' \setminus C|$ be the number of loads needed to transition from cache $C \in S$ to cache $C' \in S$, and modeling a request for a page $p_i \in P$ in step $i$ by the task

$$r_i(C) = \begin{cases} 0 & \text{if } p_i \in C \\ \infty & \text{otherwise,} \end{cases}$$

which forces $p_i \in C$ at the end of step $i$.

## 5.1 Lower bound for metrical task systems

We start our analysis of the abstract class of online problems induced by metrical task systems by giving a general lower bound for the competitive ratio of any online algorithm.

**Theorem 5.4** (Borodin et al. [1992]). *No deterministic online algorithm for metrical task systems with $N$ states has a competitive ratio smaller than $(2N-1)$.*

*Proof.* In order to derive a lower bound on the best possible competitive ratio for any fixed metric space $(S,\tau)$ of size $N>1$, we need to adopt the perspective of the adversary. An obvious candidate for an adversarial request sequence $\sigma=(r_1,\dots,r_n)$ for a given algorithm ALG presents tasks such that ALG happens to be in the worst possible state for each task. Introducing some parameter $\varepsilon>0$, we can achieve this by setting

$$r_i(s)=\begin{cases}\varepsilon, & \text{if } s=s_{i-1},\\ 0, & \text{else,}\end{cases}$$

where $s_{i-1}$ is the state of ALG at the start of step $i$.

Assume that ALG changes its state $n'\le n$ times overall, and let $(s_0=s_{i_0}),s_{i_1},\dots,s_{i_{n'}}$ be the states that ALG transitions to, in this order. Then

$$\text{ALG}(\sigma)=(n-n')\varepsilon+\sum_{j=1}^{n'}\tau(s_{i_{j-1}},s_{i_j}). \tag{5.1}$$

Observe that, by positivity of $\tau$ and $\varepsilon$, we have $\lim_{n\to\infty}\text{ALG}(\sigma)=\infty$. If $\lim_{n\to\infty}\text{OPT}(\sigma)$ is bounded, then ALG has an unbounded competitive ratio. In the following, we can therefore assume that $\lim_{n\to\infty}\text{OPT}(\sigma)=\infty$.

We bound the cost of OPT using the averaging technique. We use a class of algorithms $\mathcal{A}$ that depends on the behavior of ALG. Specifically, $\mathcal{A}$ consists of $(2N-1)$ algorithms, such that, after every step, exactly two algorithms of $\mathcal{A}$ are in each state *not* occupied by ALG and exactly one algorithm is in the same state as ALG, while at most one algorithm in $\mathcal{A}$ switches state in every step. Observe that we can achieve this by switching one algorithm from state $s_i$ to state $s_{i-1}$ whenever ALG transitions from $s_{i-1}$ to $s_i$. Let $B_0=B_0(\mathcal{M})>0$ denote the total (constant) setup cost required in step 1 to distribute the algorithms in $\mathcal{A}$ over starting states according to the system we just described.

To compute the average cost $c_{\text{avg}}(\sigma)$ of the algorithms in $\mathcal{A}$, observe that the algorithms in state $s_{i-1}$ after step $i$ have non-zero cost $\varepsilon$, while all other algorithms have cost 0. If ALG changes state away from $s_{i-1}$ in step $i$, two algorithms in $\mathcal{A}$ have cost $\varepsilon$, otherwise only one algorithm has cost $\varepsilon$. Also, whenever ALG transitions between states, so does a single algorithm in $\mathcal{A}$, otherwise there are no state transitions. Since ALG transitions exactly $n'$ times between states, we get

$$c_{\text{avg}}(\sigma)=\frac{1}{2N-1}\left[(n-n')\varepsilon+\sum_{j=1}^{n'}\tau(s_{i_j},s_{i_{j-1}})+2n'\varepsilon+B_0\right].$$

With eq. (5.1) this gives

$$\text{OPT}(\sigma)\le c_{\text{avg}}(\sigma)=\frac{2n'\varepsilon+\text{ALG}(\sigma)+B_0}{2N-1}. \tag{5.2}$$

With $\text{ALG}(\sigma)\ge\sum_{j=1}^{n'}\tau(s_{i_{j-1}},s_{i_j})\ge n'\tau_{\min}$, this yields

$$(2\varepsilon/\tau_{\min}+1)\cdot\text{ALG}(\sigma)\ge 2n'\varepsilon+\text{ALG}(\sigma)\overset{(5.2)}{\ge}(2N-1)\cdot\text{OPT}(\sigma)-B_0.$$

Observe that $\lim_{n\to\infty}\text{ALG}(\sigma)=\lim_{n\to\infty}\text{OPT}(\sigma)=\infty$. We obtain

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)}\ge\frac{2N-1}{1+2\varepsilon/\tau_{\min}}-\frac{B_0}{(1+2\varepsilon/\tau_{\min})\cdot\text{OPT}(\sigma)}\to_{\varepsilon\to0}(2N-1)-B_0/\text{OPT}(\sigma)\to_{n\to\infty}2N-1.$$

By Proposition 1.12, this implies that ALG cannot be better than $(2N-1)$-competitive for all $\varepsilon>0$. □

*Remark* 5.5. Observe that this lower bound implies that if we solve the paging problem with an algorithm for metrical task systems applied to the metrical task system of Example 5.3, we can only provide very weak bounds on the competitive ratio, since, generally, $N = \binom{|P|}{k} \gg k$.

## 5.2 Work function algorithm

We now design an online algorithm WFA for metrical task systems using the paradigm of a *work function algorithm*. The principal idea of this approach is to, in every step, explicitly compute the final state of an optimum solution for the request sequence so far, and try to stay close to this state. Obviously, we need a way to compute optimum solutions to use this scheme.

Let $w_i(s)$ be the optimum offline cost required to first process the request sequence $\sigma_{\leq i} = (r_1, \ldots, r_i)$ and then transition to state $s \in S$, and, in particular, $\text{OPT}(\sigma) = \min_{s \in S} w_n(s)$. We can compute this cost via dynamic programming:

$$\begin{aligned} w_0(s) &= \tau(s_0, s), \\ w_i(s) &= \min_{x \in S} \{w_{i-1}(x) + r_i(x) + \tau(x, s)\}. \end{aligned} \tag{5.3}$$

Ideally, we want WFA to always be in a state $s$ that minimizes $w_i(s)$. However, if $\tau(s_{i-1}, s)$ is large and $w_i(s_{i-1})$ is not much worse than $w_i(s)$, it may not be worth it to transition to $s$. Instead, we let WFA move to a state

$$s_i \in \arg\min_{x \in S} \{w_i(x) + \tau(s_{i-1}, x)\}. \tag{5.4}$$

In fact, this choice leaves some freedom. The following lemma states that we can always select $s_i$ such that there is an offline optimum for first serving the request sequence $\sigma_{\leq i}$ and then moving to $s_i$, such that the last request is already served in state $s_i$. We define WFA such that it selects $s_i$ accordingly.

**Lemma 5.6.** WFA *can select* $s_i$ *such that*

$$w_i(s_i) = w_{i-1}(s_i) + r_i(s_i).$$

*Proof.* Take any state $s \in \arg\min_{x \in S}\{w_i(x) + \tau(s_{i-1}, x)\}$ and let $x' \in \arg\min_{x \in S}\{w_{i-1}(x) + r_i(x) + \tau(x, s)\}$ be a state that minimizes eq. (5.3) for $s$, i.e.,

$$w_i(s) = w_{i-1}(x') + r_i(x') + \tau(x', s). \tag{5.5}$$

We claim that we can set $s_i = x'$. To show this, we need to show that (i) $x' \in \arg\min_{x \in S}\{w_i(x) + \tau(s_{i-1}, x)\}$ and (ii) $w_i(x') = w_{i-1}(x') + r_i(x')$.

To see (i), we observe that, by definition of $w_i$ (eq. (5.3)), we have $w_i(x') \leq w_{i-1}(x') + r_i(x')$, and thus

$$\begin{aligned} w_i(x') + \tau(s_{i-1}, x') &\overset{(5.3)}{\leq} w_{i-1}(x') + r_i(x') + \tau(s_{i-1}, x') \\ &\overset{(5.5)}{=} w_i(s) - \tau(x', s) + \tau(s_{i-1}, x') \\ &\overset{\triangle-\text{ineq.}}{\leq} w_i(s) + \tau(s_{i-1}, s). \end{aligned} \tag{5.6}$$

To see (ii), we observe that, by definition of $s$, we have $w_i(s) + \tau(s_{i-1}, s) \leq w_i(x') + \tau(s_{i-1}, x')$, and hence eq. (5.6) holds with equality. But then, all steps in the derivation of eq. (5.6) also hold with equality, and, in particular, the first step yields

$$w_i(x') = w_{i-1}(x') + r_i(x'). \qquad \square$$

Recall that the proof of Theorem 5.4 relied on a class of algorithms $\mathcal{A}$ that depends on the online algorithm. In particular, the proof used averaging, i.e., the fact that $\textsc{Opt}(\sigma)$ is bounded by the average cost over the $(2N-1)$ algorithms in $\mathcal{A}$. To get a tight upper bound for Wfa, we compare the cost incurred by Wfa to the *total* cost of the algorithms in $\mathcal{A}$, which is obviously at most $(2N-1)$ times the average cost. Since, after step $i$, $\mathcal{A}$ has exactly two algorithms in every state $s \neq s_i$ and exactly one algorithm in state $s_i$, we can compute a lower bound $B_i$ on the total cost up to step $i$ by pretending that every algorithm in $\mathcal{A}$ chose an optimal way of reaching its final state. We get

$$B_i = w_i(s_i) + 2\sum_{s \neq s_i} w_i(s). \tag{5.7}$$

We can now show that the cost of Wfa in step $i$ is bounded from above by the total cost in step $i$ of all algorithms in $\mathcal{A}$.

**Lemma 5.7.** *For all $i \in \{1, \ldots, n\}$ we have $\textsc{Wfa}_i(\sigma) \leq B_i - B_{i-1}$.*

*Proof.* The cost of Wfa in step $i$ is defined as

$$\textsc{Wfa}_i(\sigma) = \tau(s_{i-1}, s_i) + r_i(s_i).$$

We separately bound both terms. For $\tau(s_{i-1}, s_i)$, we use the definition of $s_i$ to obtain

$$\begin{aligned} w_i(s_i) + \tau(s_{i-1}, s_i) &\overset{(5.4)}{=} \min_{x \in S}\{w_i(x) + \tau(s_{i-1}, x)\} \\ &\leq w_i(s_{i-1}) + \tau(s_{i-1}, s_{i-1}) \\ &= w_i(s_{i-1}), \end{aligned}$$

which implies

$$\tau(s_{i-1}, s_i) \leq w_i(s_{i-1}) - w_i(s_i).$$

For $r_i(s_i)$, by Lemma 5.6,

$$r_i(s_i) = w_i(s_i) - w_{i-1}(s_i).$$

Together, we get

$$\textsc{Wfa}_i(\sigma) \leq w_i(s_{i-1}) - w_{i-1}(s_i). \tag{5.8}$$

On the other hand, using $w_i(s) \geq w_{i-1}(s)$, we have

$$\begin{aligned} B_i - B_{i-1} &= 2\sum_{s \neq s_i} w_i(s) + w_i(s_i) - 2\sum_{s \neq s_{i-1}} w_{i-1}(s) - w_{i-1}(s_{i-1}) \\ &= 2\sum_{s \notin \{s_i, s_{i-1}\}} (w_i(s) - w_{i-1}(s)) + 2w_i(s_{i-1}) + w_i(s_i) - 2w_{i-1}(s_i) - w_{i-1}(s_{i-1}) \\ &\geq (2w_i(s_{i-1}) - w_{i-1}(s_{i-1})) + (w_i(s_i) - 2w_{i-1}(s_i)) \\ &\geq w_i(s_{i-1}) - w_{i-1}(s_i) \\ &\overset{(5.8)}{\geq} \textsc{Wfa}_i(\sigma), \end{aligned}$$

as claimed. □

Lemma (5.7) implies that Wfa is not more expensive than the total cost of all algorithms in the class $\mathcal{A}$. This means that on the adversarial request sequence that we constructed for the lower bound, Wfa is only a factor of at most $(2N-1)$ away from the bound on the optimal offline cost we constructed via averaging. We show that this holds for all request sequences and all bounds for $\textsc{Opt}(\sigma)$.

**Theorem 5.8** (Borodin et al. [1992])**.** WFA *is* $(2N-1)$*-competitive for metrical task systems with* $N$ *states.*

*Proof.* Using Lemma 5.7, and setting $\alpha = \alpha(\mathcal{M}) := (2N-1)\max_{x,y\in S}\tau(x,y) - B_0(\mathcal{M})$, we obtain

$$\begin{aligned}
\text{WFA}(\sigma) &= \sum_{i=1}^{n} \text{WFA}_i(\sigma) \\
&\overset{\text{Lem. 5.7}}{\leq} B_n - B_0 \\
&\overset{(5.7)}{\leq} (2N-1)\max_{s\in S} w_n(s) - B_0 \\
&\overset{(5.3)}{=} (2N-1)\max_{s\in S}\min_{x\in S}(w_{n-1}(x) + r_n(x) + \tau(x,s)) - B_0 \\
&\leq (2N-1)\min_{x\in S}(w_{n-1}(x) + r_n(x)) + (2N-1)\max_{x,y\in S}\tau(x,y) - B_0 \\
&\leq (2N-1)\min_{x,s\in S}(w_{n-1}(x) + r_n(x) + \tau(x,s)) + \alpha \\
&\overset{(5.3)}{=} (2N-1)\min_{s\in S} w_n(s) + \alpha \\
&= (2N-1)\cdot \text{OPT}(\sigma) + \alpha
\end{aligned}$$

□

# 6 The $k$-Server Problem

In the last chapter, we considered metrical task systems as a broad, abstract class of online problems that feature a system state with metric transition costs between states. Aside from this restriction, the costs of a request in a metrical task system may vary arbitrarily between states. As a result, the competitive ratio for metrical task systems depends on the total number of states of the system, which only allows for very weak bounds in general. In this chapter, we consider a subclass of metrical task systems with a more restrictive cost structure that, aside from metric transition costs, only allows to distinguish between permitted and forbidden states (cost 0 or $\infty$). More specifically we assume that the system state is comprised of the individual states of $k$ servers, and a request is given by a state that must be occupied by at least one of the servers. Because of the geometric interpretation, we speak of points of the metric space instead of states.

| $k$-Server Problem | |
|---|---|
| **given:** | metric space $(P, d)$, initial server locations $S_0 = (S_{0,1}; \dots; S_{0,k}) \in P^k$ |
| **online:** | sequence $\sigma = (p_1, \dots, p_n) \in P^n$ of points |
| **actions:** | for every $j \in \{1, \dots, k\}$: move server $j$ to $S_{i,j}$ (cost: $d(S_{i-1,j}; S_{i,j})$) |
| (step $i$) | afterwards there must be a server $j$ with $S_{i,j} = p_i$ |
| **objective:** | minimize the total cost $\sum_{i=1}^{n} \textsc{Alg}_i(\sigma) = \sum_{i=1}^{n} \sum_{j=1}^{k} d(S_{i-1,j}; S_{i,j})$ |

We observe that $k$-server problems are a restriction of metrical task systems that still covers the paging problem (for example).

*Remark* 6.1. The $k$-server problem can be formulated as a metrical task system where $S_i$ is the system state after step $i$ and a request for point $p_i$ is translated into a cost function that assigns $S_i$ a cost of 0 if there is a server $j$ with $S_{i,j} = p_i$ and $\infty$ otherwise. With this formulation, Theorem 5.8 gives us an exponential upper bound of $(2N^k - 1)$ on the competitive ratio for the $k$-server problem for metric spaces with $N$ points. In the following, we derive better bounds tailored to the $k$-server problem.

**Example 6.2.** The $k$-server problem is still abstract enough to encompass the ($k$-)paging problem. We can see this by letting the set of points $P$ be identical to the set of pages and interpreting the server locations as the set of pages that are in the cache. To obtain cost 1 for every page load, we simply set the distance between any pair of points/pages to 1.

*Remark* 6.3. Recall that for the paging problem we could restrict ourselves to demand paging algorithms that only evict/load (at most) one page at a time, and only upon a page fault. Very similarly, for the $k$-server problem, we can restrict ourselves to *lazy* algorithms that, in step $i$, only move the server that serves request $p_i$. We can transform any algorithm to be lazy as follows: If a server $s$ moves without serving a request, we instead keep the server stationary and only pretend that the server moved, and remember that we still need to execute the move later. Once server $s$ needs to move to serve a request, we first execute all remembered moves, which brings the server to its supposed location, and then we execute the move to serve the request. The cost of the modified algorithm may be smaller than that of the original algorithm, but not larger.

A straight-forward lazy algorithm for the $k$-server problem is the Greedy algorithm that always moves the server that is closest to the requested point with respect to the distance function $d$. To see that this algorithm is not competitive, consider a metric space with three points $P = \{a, b, c\}$ where $d(a, b) < \min\{d(a, c), d(b, c)\}$, and two servers initially at $S_0 = (c, a)$ (see Figure 7). For the request sequence $\sigma = (b, a, b, a, b, a, \dots) \in P^n$, Greedy keeps one of the servers at $c$ forever and incurs a cost of $\textsc{Greedy}(\sigma) = n \cdot d(a, b)$, while $\textsc{Opt}(\sigma) = d(c, b)$ is independent of $n$.

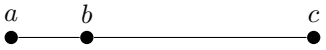


Figure 7: Linear embedding of the bad example for GREEDY with two servers initially at $S_0 = (a, c)$ and the request sequence $\sigma = (b, a, b, a, \dots)$.

## 6.1 Lower bound for the $k$-server problem

The $k$-server problem becomes trivial for finite metric spaces with $|P| = k$ once servers have been distributed. We now show that as soon as $P$ contains more than $k$ points this is no longer the case, independent of the metric space.

**Theorem 6.4** (Manasse et al. [1990])**.** *No deterministic online $k$-server algorithm has a competitive ratio lower than $k$ on **any** metric space $(P, d)$ with at least $k + 1$ points.*

*Proof.* For any fixed (lazy) online algorithm ALG and any metric space $(P, d)$ with at least $k + 1$ points, we construct an adversarial request sequence $\sigma = (p_1, \dots, p_n) \in P^n$ for ALG. Without loss of generality, we may assume that $P$ consists of exactly $k+1$ points, since we can simply ignore any additional point. A natural idea is to choose start with servers distributed over $k$ distinct points of $P$, and let $\sigma$ always request the unique point not occupied by ALG. Since ALG is lazy, in step $i < n$, it moves exactly one server, and, by construction of $\sigma$, this move is from $p_{i+1}$ to $p_i$. Let $p_{n+1}$ denote the position of the server that serves request $p_n$ at the start of step $n$. Then,

$$\text{ALG}(\sigma) = \sum_{i=1}^{n} d(p_{i+1}, p_i), \tag{6.1}$$

and, in particular, by positivity of $d$, we have $\lim_{n\to\infty} \text{ALG}(\sigma) = \infty$.

We derive a bound on $\text{OPT}(\sigma)$ using the averaging technique. For this, we need to define a class $\mathcal{A}$ of simple algorithms, i.e., simple rules which server to select for serving request $p_i$. If there is a server at $p_i$, we obviously use this server. Otherwise, $p_i$ is the only unoccupied point. The most natural candidate to serve $p_i$ is the server at the closest point to $p_i$, but this gives the GREEDY algorithm, which we already showed to be bad. Another server that stands out is the server at $p_{i-1}$ (for $i > 1$). In order to define a class of algorithms that follow the simple rule of serving each request for an unoccupied point $p_i$ with the server at $p_{i-1}$, we define an algorithm for every possible set $S_1$ of (distinct) positions of all servers at the end of step 1. Of course, we must have $p_1 = S_{1,j}$ for some $j \in \{1, \dots, k\}$, therefore, since $|P| = k + 1$, there are exactly $\binom{|P\setminus\{p_1\}|}{k-1} = k$ different starting configurations (we do not distinguish individual servers). The average cost to put the algorithms in their configuration in step 1 is a constant $\alpha = \alpha(P, d) \le \max_{p,q\in P} d(p, q)$. Note that the algorithms in $\mathcal{A}$ are not lazy (in step 1), but could be transformed to be lazy.

We claim that no two algorithms can ever be in the same configuration. This is true initially for the request sequence introduced above. If it is the case until step $i - 1$, then, at the start of step $i \ge 2$, there is exactly one algorithm in $\mathcal{A}$ that does not have a server at $p_i$ already, and all algorithms have a server at $p_{i-1}$. The only algorithm that needs to move a server in step $i$ (from $p_{i-1}$ to $p_i$) is the only one without a server at position $p_{i-1}$ after step $i$. Our claim follows by induction.

Since exactly one algorithm needs to move in every step, the total cost of the algorithms in $\mathcal{A}$ is $k\alpha + \sum_{i=2}^{n} d(p_{i-1}, p_i)$. The optimum cost is bounded by the average cost $c_{\text{avg}} = \alpha + \frac{1}{k}\sum_{i=2}^{n} d(p_{i-1}, p_i)$. With eq. (6.1), and by positivity of $d$, we get

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} \ge \frac{\sum_{i=1}^{n} d(p_i, p_{i+1})}{\alpha + \frac{1}{k}\sum_{i=2}^{n} d(p_{i-1}, p_i)} \ge \frac{k}{1 + k\alpha/\sum_{i=2}^{n} d(p_{i-1}, p_i)} \to_{n\to\infty} k.$$

Using $\lim_{n\to\infty} \text{ALG}(\sigma) = \infty$, Proposition 1.12 thus excludes any competitive ratio below $k$. □

Based on this lower bound and matching upper bounds for some special cases that we will see below, Manasse, McGeoch and Sleator posed their famous "$k$-server conjecture":

**Conjecture 6.5** (Manasse et al. [1990]). *There is a deterministic $k$-competitive $k$-server algorithm.*

Proving or disproving this conjecture for general metric spaces is one of the most important open problems in online optimization. In the following sections, we prove the $k$-server server conjecture for specific metric spaces.

## 6.2 The $k$-server problem on the line

We first consider the simple metric space $(P, d)$ of the *(Euclidean) line*, i.e., with $P = \mathbb{R}$ and where $d$ is the Euclidean distance between points. Recall the bad instance for the Greedy algorithm we discussed above (Figure 7). Since this example is already embedded on the line, this means that we need to be more careful how to decide which server to move. If $d(b, c) \gg d(a, b)$, the first requests must be served by the server at $a$, otherwise we are not competitive on short sequences. But at some point we have to involve the server at $c$ to avoid an unbounded cost compared to the optimum as instances grow.

One (non-lazy) idea how to improve Greedy's behavior in the example of Figure 7 would be to move both servers towards a request at $b$ and let the server serve the request that first reaches it. The algorithm Dc (*double cover*) generalizes this idea to any number of servers by moving a closest server towards request $p_i$ from both sides if such a server exists. We call the two servers closest to $p_i$ from either side its *neighbors*, and if there are multiple closest servers on one side, we pick only one of them arbitrarily as the neighbor of $p_i$. Then, in step $i$, Dc moves the (at most two) neighbors of $p_i$ towards $p_i$ with the same speed until (at least) one of them reaches $p_i$. Observe that Dc behaves as intended on the example of Figure 7: The first requests are served by the server starting at $a$, and at some point the server that started at $c$ reaches $b$ and stays there. Eventually, the servers are located at $a$ and $b$ and do not need to move anymore.

We analyze Dc using the potential function method. Recall that the principal idea of defining a potential function is to allow investing additional cost in cheap steps in order to pay for expensive steps. One reason why a $k$-server request on the line can be costly for Dc, in terms of relative cost compared to Opt, is that the server positions of Dc and Opt are very different. We can measure this by the cost $M_i(\sigma)$ of a minimum (distance) matching between the server positions of Dc and Opt at the end of step $i$. A reason why a request can be costly in terms of absolute cost is that Dc's servers are far apart. We can measure this by the sum $\Sigma_i(\sigma) := \sum_{j \neq j'} d(S_{i,j}, S_{i,j'})$ of pairwise distances of Dc's servers at the end of step $i$. With this intuition, we define a potential function of the form

$$\Phi_i(\sigma) = a_1 M_i(\sigma) + a_2 \Sigma_i(\sigma),$$

with some parameters $a_1, a_2 \geq 0$.

We want to use the alternating moves approach (Corollary 4.3), i.e., we pretend that Opt moves before Dc in step $i$ and let $\Phi'_i(\sigma)$ denote the intermediate potential function value after Opt moved. To show that Dc is $\rho$-competitive, it is sufficient to show (i) $\Delta\Phi_i^{\text{Opt}} := \Phi'_i(\sigma) - \Phi_{i-1}(\sigma) \leq \rho \cdot \text{Opt}_i(\sigma)$, (ii) $\Delta\Phi_i^{\text{Dc}} := \Phi_i(\sigma) - \Phi'_i(\sigma) \leq -\text{Dc}_i(\sigma)$, (iii) $\Phi_i(\sigma) \geq 0$, and (iv) there is a constant $\beta \geq 0$ independent of $\sigma$ with $\Phi_0(\sigma) \leq \beta$. Then all conditions of Corollary 4.3 are met, since the amortized cost is bounded by

$$a_i(\sigma) = \text{Dc}_i(\sigma) + \Delta\Phi_i = \text{Dc}_i(\sigma) + \Delta\Phi_i^{\text{Opt}} + \Delta\Phi_i^{\text{Dc}} \leq \rho \cdot \text{Opt}_i(\sigma).$$

Properties (iii) and (iv) hold as long as $S_0 \in P^k$ is fixed, because $\Phi_i \geq 0$ and $\Phi_0 \leq \beta(S_0) := \sum_{j \neq j'} d(S_{0,j}, S_{0,j'})$.

For Property (i), we may assume that Opt is lazy, i.e., that it moves at most one server in every step $i$. Assume that Opt moves a server by $\text{Opt}_i(\sigma) = d \geq 0$. This move may increase $M_i(\sigma)$ by at most $d$, even if we maintain the same (not necessarily minimum) matching. Then Property (i) holds if $\Delta\Phi_i^{\text{Opt}} \leq a_1 d \leq \rho \cdot \text{Opt}_i(\sigma) = \rho d$, i.e., if $a_1 \leq \rho$. For Property (ii), we distinguish two cases in step $i$.

Case 1: DC moves a single server by $d$. Then, by definition of DC, this server must lie on the boundary of the convex hull of all server positions, and the requested point must lie outside of the convex hull. This means that the server moves away from all other servers and $\Sigma_i(\sigma)$ increases by exactly $(k-1)d$. Also, there is a minimum matching that matches the server of DC at $p_i$ after the move to the server at $p_i$ in OPT. This must also have been a minimum matching before the move, since there are no servers closer to $p_i$ in DC. Therefore $M_i(\sigma)$ is decreased by exactly $d$. Property (ii) holds in this case if $\Delta\Phi_i^{\text{DC}} = a_2(k-1)d - a_1 d \leq -\text{DC}_i(\sigma) = -d$, i.e., if $a_1 \geq a_2(k-1)+1$.

Case 2: DC moves two servers by $d$ each. Then, by definition of DC, the two servers $j, j' \in \{1,\ldots,k\}$ move towards each other with no other servers between them. This means that the contribution of the pair $(j,j')$ to $\Sigma_i$ decreases by exactly $2d$, and the changes in distance from $j$ to every other server cancels with the change in distance from $j'$. After the move, there is a minimum matching that matches $j$ or $j'$ to the server of OPT at $p_i$. This must also have been a minimum matching before the move, since there are no servers between $j$ and $j'$. Consequently, $M_i(\sigma)$ does not increase during the move. Property (ii) holds in this case if $\Delta\Phi_i^{\text{DC}} \leq -a_2 2d \leq -2d = -\text{DC}_i(\sigma)$, i.e., if $a_2 \geq 1$.

Overall, all properties hold if

$$\rho \geq a_1 \geq a_2(k-1)+1 \geq (k-1)+1 = k,$$

and we get equality for $a_1 = \rho = k$ and $a_2 = 1$, i.e., for the potential $\Phi_i(\sigma) = kM_i(\sigma) + \Sigma_i(\sigma)$.

**Theorem 6.6** (Chrobak et al. [1991])**.** DC *is $k$-competitive for the $k$-server problem on the line if $\sum_{j\neq j'} d(S_{0,j}, S_{0,j'})$ can be bounded by a constant.*

### 6.2.1 Double Cover on trees

There is a direct generalization for DC from the real line to the metric space induced by a tree $T = (V,E)$ with edge lengths, where the distance between two vertices is defined by the metric closure (i.e., the shortest path distance). Let $v \in V$ be a vertex of $T$, consider a server $j$ located at a vertex $v_j$, and let $P$ be the unique $v_j$-$v$-path in $T$. If there is no other server located along $P \setminus \{v_j\}$ and if there is no server $j' \in \{1,\ldots,j-1\}$ located at $v_j$, we call $j$ a *neighbor* of $v$. We generalize DC by continuously moving *all* neighbors of $v$ towards each request for a vertex $v \in V$, until the first server reaches $v$. Note that the set of neighbors can become smaller over time. Also note that, since the vertices of $T$ are the points of the metric space, we cannot continuously move servers along edges. Similarly to the transformation into a lazy algorithm, we can avoid this issue by executing all movements of a server only virtually, until the server actually serves a request at a vertex.

**Theorem 6.7** (Chrobak and Larmore [1991b])**.** DC *is $k$-competitive for the $k$-server problem on a tree if $\sum_{j\neq j'} d(S_{0,j}, S_{0,j'})$ can be bounded by a constant..*

*Proof.* We use the same potential as in the proof of Theorem 6.6, i.e.,

$$\Phi_i(\sigma) = kM_i(\sigma) + \Sigma_i(\sigma).$$

As before, we can still assume OPT to be lazy, and therefore $\Delta\Phi_i^{\text{OPT}} \leq kd = k \cdot \text{OPT}_i(\sigma)$.

Now subdivide DC's move in step $i$ into time periods $(t_1,t_2],(t_2,t_3],\ldots,(t_{N-1},t_N]$, such that the set of servers that move stays the same during each period. Consider a period $(t_j,t_{j+1}]$ where $m_j$ servers move by $d_j$ each. Observe that, during the whole period, there is a minimum matching where one the moving servers is matched to OPT's server at the requested vertex $v_i \in V$. This means that $M_i$ can increase by at most $(m_j-1)d_j - d_j = m_j d_j - 2d_j$, which causes an increase in potential by at most $km_j d_j - 2kd_j$. Each stationary server gets closer to at least $m_j - 1$ other servers and further away from at most one other server. The contribution to $\Sigma_i$ of all these changes in pairwise distance between one of the $k - m_j$ stationary servers and a moving server sum to

$$-(k-m_j)((m_j-1)d_j - d_j) = -(k-m_j)(m_j d_j - 2d_j).$$

Finally, the pairwise distances between moving servers each decrease by exactly $2d_j$, which changes $\Sigma_i$ by $-\frac{m_j(m_j-1)}{2} \cdot 2d_j = -m_j(d_j m_j - d_j)$.
Overall, the change in potential during one period is at most

$$km_j d_j - 2kd_j - (k - m_j)(m_j d_j - 2d_j) - m_j(d_j m_j - d_j) = -m_j d_j,$$

which corresponds exactly to the total distance traveled in the period. Summed over all periods in step $i$, we get

$$\Delta\Phi_i^{\text{DC}} = -\sum_{j=1}^{N-1} m_j d_j = -\text{DC}_i(\sigma).$$

Together with our bound on $\Delta\Phi_i^{\text{OPT}}$ this gives an amortized cost of

$$a_i(\sigma) = \text{DC}_i(\sigma) + \Delta\Phi_i(\sigma) = k \cdot \text{OPT}_i(\sigma).$$

We also have $\Phi_i \geq 0$ and $\Phi_0 \leq \frac{k(k-1)}{2} \max_{v,v' \in V} d(v, v')$, which is constant if the tree is fixed (and finite). We can therefore apply the potential function method (Corollary 4.3) to prove the theorem. □

*Remark* 6.8. Note that the ($k$-)paging problem can be modeled as the $k$-server problem on a star-shaped tree with one vertex per memory page and one server per cache slot (see Figure 8). The positions of the servers correspond to the pages in the cache, evicting a page corresponds to moving the respective server to the center node, and loading a page corresponds to moving a server to the respective node. Note that in our model, costs are split between loading and evicting, but this makes no difference for demand paging algorithms, provided that the cache is full initially. On the star associated with a paging instance, DC becomes equivalent to the paging algorithm FWF. This means that $k$-competitiveness of DC on trees implies $k$-competitiveness of FWF for paging. It is worth emphasizing that the $k$-server framework captures the paging problem tightly enough to allow tight bounds to be derived via the much more general setting.

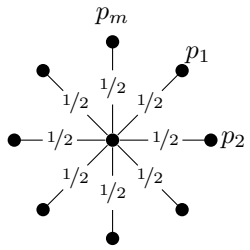


Figure 8: Illustration of the ($k$-)paging problem expressed as a $k$-server problem.

### 6.2.2 Double Cover on finite metric spaces

We can further generalize DC to any finite metric space with $N$ points, by interpreting such a space as a complete graph $G = (V, E)$ with a vertex for every point of the space and edge lengths that correspond to the distances between every pair of points. We can apply our adaptation of DC for trees by restricting ourselves to the minimum spanning tree $T$ of $G$. This means that we forget all edges not in $T$ and serve all requests by only moving along $T$. By definition of minimum spanning trees, every edge $e$ of $G \setminus T$ closes a cycle $C$ in $T$, and $e$ must be a longest edge of $C$. If $e$ has length $d$, this means that $C \setminus \{e\}$ has length at most $(N-1)d$, i.e., there is a detour in $T$ for $e$ that has length at most $(N-1)d$. Applying this argument multiple times yields that every shortest path of length $d'$ in $G$ has a detour of length at most $(N-1)d'$ in $T$.

Let $\text{OPT}_{\text{tree}}$ be the offline optimum cost when restricted to moving along edges of $T$. Then, by Theorem 6.7, there is $\alpha \in \mathbb{R}$ with

$$\text{DC}(\sigma) \leq k \cdot \text{OPT}_{\text{tree}}(\sigma) + \alpha \leq k(N-1) \cdot \text{OPT}(\sigma) + \alpha,$$

and we obtain the following.

**Theorem 6.9** (Chrobak and Larmore [1991b])**.** DC *is $(Nk-k)$-competitive for the $k$-server problem on any finite metric space with $N$ points if $\sum_{j \neq j'} d(S_{0,j}, S_{0,j'})$ can be bounded by a constant..*

Note that this is much better than the competitive ratio of $2N^k - 1$ that follows from the fact that the $k$-server problem is a metrical task system.

## 6.3 Balancing algorithm*

For arbitrary metric spaces, one of the most natural online algorithms is the algorithm BAL that greedily balances the distances moved by the servers. Let $D_j^i = \sum_{i'=1}^{i} d(S_{i'-1,j}, S_{i',j})$ denote the total distance that server $j$ moved until the end of step $i$. Then BAL serves the request $p_i$ in step $i$ with any server $j \in \arg\min_{j' \in \{1,\dots,k\}} \{D_{j'}^{i-1} + d(S_{i-1,j'}, p_i)\}$ that is closest to $p_i$. In particular, if there is a server $j$ at $p_i$, this server is selected, since $d(S_{i-1,j}, p_i) = 0$ and $D_j^{i-1} \leq D_{j'}^{i-1} + d(S_{i-1,j'}, p_i)$ for every $j' \in \{1, \dots, k\}$, because either $D_j^{i-1} = 0$ or because of the fact that $j$ was selected earlier to move to $p_i$. In particular, if servers start at distinct locations, they always remain at distinct locations. In the following, we assume that this is the case.

**Proposition 6.10.** *For every $k \geq 2$,* BAL *is not competitive for the $k$-server problem on metric spaces of size $N \geq k + 2$.*

*Proof.* Consider the example in Figure 9 with $k = 2$ servers and $N = 4$ points. We define the adversarial input sequence $\sigma = (a, b, c, d, a, b, c, d, \dots) \in P^n$. For this sequence $\text{BAL}(\sigma) = n$, since one server serves all requests at $\{a, d\}$ and the other serves the requests at $\{b, c\}$. On the other hand $\text{OPT}(\sigma) = 1 + (n-1)\varepsilon$, since one server serves all requests at $\{a, b\}$ and the other serves the requests at $\{c, d\}$. We immediately get

$$\frac{\text{BAL}(\sigma)}{\text{OPT}(\sigma)} = \frac{n}{1 + (n-1)\varepsilon} \to_{\varepsilon \to 0} n.$$

Note that the example can be extended to larger $k, N$ simply by adding additional points and servers far away from $\{a, b, c, d\}$. □

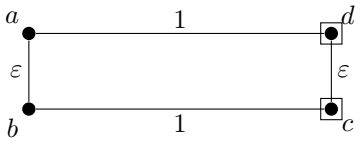


Figure 9: Adversarial instance of Proposition 6.10 with points $P = \{a, b, c, d\}$ and initial server locations $S_0 = (c, d)$. The distances obey $d(a, b) = d(c, d) = \varepsilon \ll 1$.

*Remark* 6.11. Irani and Rubinfeld [1991] showed that minimizing $D_j^{i-1} + 2d(S_{i-1,j}, p_i)$ yields a 10-competitive 2-server algorithm, and Chrobak and Larmore [1991a] showed that this algorithm has competitive ratio at least 6. Kleinberg [1994] showed that minimizing $D_s^{i-1} + f(d(s, p_i))$ for any function $f : \mathbb{R} \to \mathbb{R}$ does not yield a 2-server algorithm with competitive ratio smaller than $\frac{5+\sqrt{7}}{2} \approx 3.82$.

In the following, we want to show that BAL is $k$-competitive on metric spaces with $N = k + 1$ points. Since it does not make sense for the adversary to make requests that do not incur a cost for

BAL, we may assume that in every step the unique unoccupied location of BAL is requested. Let $q_i$ be the unique unoccupied point at the end of step $i \geq 0$, i.e., $q_{i-1} := p_i$ for all $i \geq 1$. Similarly, let $q_i^\star$ be an unoccupied point after step $i$ in OPT$[\sigma]$. For every point $p \in P$ that is occupied by a server $j \in \{1, \ldots, k\}$ in BAL at the end of step $i$, we define $D_p^i := D_j^i$ to be the total distance travelled by the server at $p$. Finally, let $S_{i,j}^\star$ denote the location of server $j \in \{1, \ldots, k\}$ in OPT$[\sigma]$ at the end of step $i$.

We want to apply the potential function method with the alternating moves approach (Corollary 4.3), pretending that OPT moves before BAL in every step. We introduce a rather complicated potential function, which will turn out to work well later:

$$\Phi_i(\sigma) := \begin{cases} kD_{q_i^\star}^i - \sum_{j=1}^k D_j^i & \text{if } q_i^\star \neq q_i = p_{i+1}, \\ k(D_{p_i}^i - d(p_i, p_{i+1})) - \sum_{j=1}^k D_j^i & \text{if } q_i^\star = q_i = p_{i+1}. \end{cases} \tag{6.2}$$

This potential function distinguishes between whether OPT and BAL occupy the same points ($q_i^\star = q_i$) or not ($q_i^\star \neq q_i$). At the beginning of step $i$, BAL has point $q_{i-1} = p_i$ unoccupied. After OPT moved, it must have a server at $q_{i-1} = p_i$, i.e., we are in the first case of the potential function. Since BAL's servers did not move yet, the intermediate potential value is

$$\Phi_i'(\sigma) := kD_{q_i^\star}^{i-1} - \sum_{j=1}^k D_j^{i-1}.$$

In order to apply Corollary 4.3, we have to ensure that conditions *(i)-(iv)* are satisfied. For condition *(i)*, it is sufficient to show that there is a constant $\alpha(\mathcal{M}) \geq 0$ such that for all $j \in \{1, \ldots, k\}$ we have $kD_j^i - \sum_{j'=1}^k D_{j'}^i \geq -\alpha(\mathcal{M})$, where $\alpha(\mathcal{M})$ only depends on the metric space $\mathcal{M} = (P, d)$. Then $\Phi_i(\sigma) \geq -kd_{\max}(\mathcal{M}) - \alpha(\mathcal{M})$ where $d_{\max}(\mathcal{M}) := \max_{p,p' \in P} d(p, p')$ only depends on $\mathcal{M}$, and condition *(i)* holds. To see this, we may individually bound differences of the form $\left|D_{j'}^i - D_j^i\right|$ for $j, j' \in \{1, \ldots, k\}$, because we need an upper bound for

$$\sum_{j'=1}^k D_{j'}^i - kD_j^i = \sum_{j'=1}^k (D_{j'}^i - D_j^i) \leq \sum_{j'=1}^k |D_{j'}^i - D_j^i|.$$

*Claim* 6.12. For all $j, j' \in \{1, \ldots, k\}$ we have $|D_{j'}^i - D_j^i| \leq d(S_{i,j}, S_{i,j'})$.

With this claim, we can set $\alpha(\mathcal{M}) := k \cdot d_{\max}(\mathcal{M})$ to fulfill condition *(i)*, since then $\sum_{j'=1}^k D_{j'}^i - kD_j^i \leq \sum_{j'=1}^k d(S_{i,j}, S_{i,j'}) \leq k \cdot d_{\max}(\mathcal{M})$ and, thus, $\Phi_i(\sigma) \geq -2kd_{\max}(\mathcal{M})$.

*Proof of Claim (6.12).* We prove the claim by induction on $i$. The induction base for $i = 0$ holds, since $|D_{j'}^0 - D_j^0| = 0 \leq d(S_{i,j}, S_{i,j'})$. For the inductive step, assume that $|D_{j'}^{i-1} - D_j^{i-1}| \leq d(S_{i-1,j}, S_{i-1,j'})$ holds. The claim trivially holds if $j, j'$ do not move in step $i$. Otherwise, without loss of generality, assume that $j$ moves in step $i$, i.e., $j$ serves the request at $p_i = S_{i,j}$. We need to show that

$$|D_{j'}^i - D_j^i| = |D_{j'}^{i-1} - (D_j^{i-1} + d(S_{i-1,j}, p_i))| \leq d(p_i, S_{i,j'}) = d(p_i, S_{i-1,j'}). \tag{6.3}$$

First, by induction and triangle inequality, we have

$$D_{j'}^{i-1} - D_j^{i-1} - d(S_{i-1,j}, p_i) \overset{\text{ind.}}{\leq} d(S_{i-1,j'}, S_{i-1,j}) - d(S_{i-1,j}, p_i) \overset{\triangle-\text{ineq.}}{\leq} d(p_i, S_{i-1,j'}). \tag{6.4}$$

Now, by definition of BAL, since BAL selected server $j$ to serve $p_i$, we have $D_j^{i-1} + d(S_{i-1,j}, p_i) \leq D_{j'}^{i-1} + d(S_{i-1,j'}, p_i)$, and thus

$$D_j^{i-1} + d(S_{i-1,j}, p_i) - D_{j'}^{i-1} \overset{\text{BAL}}{\leq} D_{j'}^{i-1} + d(S_{i-1,j'}, p_i) - D_{j'}^{i-1} = d(S_{i-1,j'}, p_i). \tag{6.5}$$

Together, eqs. (6.4) and (6.5) imply eq. (6.3). □

For condition *(ii)* of Corollary 4.3, we simply observe that $\Phi_0 \leq 0$.

For condition *(iii)*, we need that $\Delta\Phi_i^{\text{OPT}} := \Phi_i' - \Phi_{i-1} \leq k \cdot \text{OPT}_i(\sigma)$. If OPT does not move in step $i$, then $q_{i-1}^\star = q_i^\star \notin \{p_{i-1}, p_i\}$, and therefore $\Delta\Phi_i^{\text{OPT}} = 0$. Otherwise, by definition, OPT moves a server from $q_i^\star \neq p_i$ to $p_i = q_{i-1}^\star$, and $\text{OPT}_i(\sigma) = d(q_i^\star, p_i)$. Because $q_{i-1}^\star = p_i$, the potential $\Phi_{i-1}$ is given by the second case of eq. (6.2) and we get

$$\begin{aligned}\Delta\Phi_i^{\text{OPT}} &= \Phi_i' - \Phi_{i-1} \\ &= (kD_{q_i^\star}^{i-1} - \sum_{j=1}^{k} D_j^{i-1}) - (k(D_{p_{i-1}}^{i-1} - d(p_{i-1}, p_i)) - \sum_{j=1}^{k} D_j^{i-1}) \\ &= k(D_{q_i^\star}^{i-1} - D_{p_{i-1}}^{i-1} + d(p_{i-1}, p_i)).\end{aligned}$$

If $q_i^\star = p_{i-1}$, it immediately follows that $\Delta\Phi_i^{\text{OPT}} = kd(p_{i-1}, p_i) = \text{OPT}_i(\sigma)$. Otherwise, in step $i-1$, BAL moved a server from $p_i \neq q_i^\star$ to $p_{i-1} \neq q_i^\star$. This means that there is a server at $q_i^\star$ that does not move in step $i$. We have

$$\begin{aligned}D_{q_i^\star}^{i-1} &= D_{q_i^\star}^{i-2} \\ D_{p_{i-1}}^{i-1} &= D_{p_i}^{i-2} + d(p_i, p_{i-1}).\end{aligned}$$

By Claim 6.12, we have $D_{q_i^\star}^{i-2} - D_{p_i}^{i-2} \leq d(q_i^\star, p_i)$, and, therefore,

$$\begin{aligned}\Delta\Phi_i^{\text{OPT}} &= k(D_{q_i^\star}^{i-1} - D_{p_{i-1}}^{i-1} + d(p_{i-1}, p_i)) \\ &= k(D_{q_i^\star}^{i-2} - D_{p_i}^{i-2}) \\ &\leq kd(p_i, q_i^\star) \\ &= k\text{OPT}_i(\sigma),\end{aligned}$$

as required by condition *(iii)*.

Finally, condition *(iv)* of Corollary 4.3 demands that $\Delta\Phi_i^{\text{BAL}} := \Phi_i - \Phi_i' \leq -\text{BAL}_i(\sigma)$, where $\text{BAL}_i(\sigma) = d(p_{i+1}, p_i)$. If $q_i^\star \neq q_i = p_{i+1}$, since also $q_i^\star \neq p_i$, BAL has a server at $q_i^\star$ that does not move during step $i$, i.e., $D_{q_i^\star}^i = D_{q_i^\star}^{i-1}$. The potential $\Phi_i$ is given by the first case of eq. (6.2) and we get

$$\begin{aligned}\Delta\Phi_i^{\text{BAL}} &= (kD_{q_i^\star}^i - \sum_{j=1}^{k} D_j^i) - (kD_{q_i^\star}^{i-1} - \sum_{j=1}^{k} D_j^{i-1}) \\ &= \sum_{j=1}^{k} D_j^{i-1} - \sum_{j=1}^{k} D_j^i \\ &= -d(p_{i+1}, p_i).\end{aligned}$$

Otherwise, we have $q_i^\star = q_i = p_{i+1}$, and $D_{p_i}^i - D_{q_i^\star}^{i-1} = D_{p_i}^i - D_{p_{i+1}}^{i-1} = d(p_i, p_{i+1})$. The potential $\Phi_i$ is given by the second case of eq. (6.2) and we get

$$\begin{aligned}\Delta\Phi_i^{\text{BAL}} &= (k(D_{p_i}^i - d(p_i, p_{i+1})) - \sum_{j=1}^{k} D_j^i) - (kD_{q_i^\star}^{i-1} - \sum_{j=1}^{k} D_j^{i-1}) \\ &= k(D_{p_i}^i - D_{q_i^\star}^{i-1} - d(p_i, p_{i+1})) - d(p_{i+1}, p_i) \\ &= -d(p_{i+1}, p_i).\end{aligned}$$

In both case condition *(iv)* is fulfilled. Using conditions *(i)-(iv)*, we can apply Corollary 4.3 and obtain the following result.

**Theorem 6.13** (Manasse et al. [1990])**.** BAL *is $k$-competitive for the $k$-server problem on finite metric spaces of size $N = k + 1$ if the initial server locations are distinct.*

## 6.4 General metric spaces

We now briefly outline additional results for general metric spaces that rely on work function approaches, i.e., on online algorithms that take the optimum behavior into account for their decisions.

For 2 servers, Manasse et al. [1990] define the algorithm RES (*residue*) that, in step $i$, serves the request at $p_i$ with the server that served the last request (at $p_{i-1}$) if and only if it can afford to do so while staying 2-competitive, even if it later needs to undo the move from $p_{i-1}$ to $p_i$ and instead move the other server to $p_i$ via $p_{i-1}$. Formally, we can express this condition via

$$2\text{OPT}(\sigma_{\leq i}) - \text{RES}(\sigma_{\leq i-1}) \geq 2d(p_{i-1}, p_i) + d(S_{i-1,1}, S_{i-1,2}),$$

where the left hand side is the so-called *residue*, i.e., the cost that RES can cause in step $i$ while still staying 2-competitive.

**Theorem 6.14** (Manasse et al. [1990])**.** RES *is 2-competitive for the online 2-server problem.*

For arbitrary $k \in \mathbb{N}$ we can define a work function algorithm much like for metrical task systems. Assuming that servers start in distinct locations, we use the fact that there is an optimum solution where servers always remain at distinct locations. We define the work function $w_i(C)$ that maps a configuration $C \subset P$, $|C| = k$ of server locations to the offline optimum cost for serving all requests in $\sigma_{\leq i}$ and then positioning the servers on the points in $C$. Let $d(C, C')$ denote the total distance of a minimum distance matching between the points in $C$ and $C'$, and let $C_0 := \{S_{0,j} : j \in \{1, \ldots, k\}\}$ denote the initial configuration of servers. We get

$$\begin{aligned} w_0(C) &= d(C_0, C) \\ w_i(C) &= \min_{p \in C}\{w_{i-1}((C \setminus \{p\}) \cup \{p_i\}) + d(p_i, p)\}. \end{aligned}$$

With this, the straight-forward work function algorithm WFA serves the request $p_i$ in step $i$ with a server

$$j \in \operatorname*{arg\,min}_{j \in \{1,\ldots,k\}} \{w(C_{i-1} \setminus \{S_{i-1,j}\} \cup \{p_i\}) + d(S_{i-1,j}, p_i)\},$$

where $C_i$ denotes the server configuration after step $i$. We state the following theorem without proof.

**Theorem 6.15** (Koutsoupias and Papadimitriou [1995])**.** WFA *is* $(2k-1)$*-competitive for the online* $k$*-server problem.*

# 7 Primal-Dual Algorithms

All underlying offline problems we have discussed in the previous chapters can be expressed in terms of an integer linear program (ILP), i.e., in the form

$$\text{(ILP)} \quad \boxed{\begin{array}{ll} \min & \boldsymbol{c}^\mathsf{T}\boldsymbol{x} \\ \text{s.t.} & A\boldsymbol{x} \geq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{N}^n \end{array}}$$

where $A \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$ and $\boldsymbol{c} \in \mathbb{R}^n$ are input parameters, and $\boldsymbol{x} \in \mathbb{R}^n$ are the solution variables.

For example, we can express the ski rental problem with $n$ days and buying price $B$ by introducing a variable $x$ that encodes whether we decide to buy skis, and variables $z_i$ that encode whether we rent skis on day $i$. We obtain the following ILP:

$$\text{(SR)} \quad \boxed{\begin{array}{lll} \min & B \cdot x + \sum_{i=1}^{n} z_i & \\ \text{s.t.} & x + z_j \geq 1, & \forall j \in \{1, \dots, n\} \\ & x, z_i \in \{0, 1\}, & \forall i \in \{1, \dots, n\} \end{array}}$$

We can turn the problem of finding a good solution to an ILP to an online problem by letting variables and constraints appear over time, while requiring monotonicity in the sense that variables that have previously been set may not be decreased.

| Online ILP Problem | |
|---|---|
| **given:** | linear program (ILP) $(A_0, \boldsymbol{b}_0, \boldsymbol{c}_0)$, initial solution $\boldsymbol{x}_0$ |
| **online:** | sequence $\sigma = ((A_1, \boldsymbol{b}_1, \boldsymbol{c}_1), (A_2, \boldsymbol{b}_2, \boldsymbol{c}_2), \dots, (A_n, \boldsymbol{b}_n, \boldsymbol{c}_n))$ of ILPs, s.t. $A_i, \boldsymbol{b}_i, \boldsymbol{c}_i$ extend $A_{i-1}, \boldsymbol{b}_{i-1}, \boldsymbol{c}_{i-1}$ by adding rows & columns |
| **actions:** | extend solution $\boldsymbol{x}_{i-1}$ to a new feasible solution $\boldsymbol{x}_i$ for $A_i, \boldsymbol{b}_i, \boldsymbol{c}_i$ |
| (step $i$) | by setting all new variables and by increasing or keeping old variables |
| **objective:** | minimize $\boldsymbol{c}_n^\mathsf{T}\boldsymbol{x}_n$ |

This view on online problems will allow us to employ powerful linear programming methods to develop and analyze online algorithms.

## 7.1 The primal-dual method

Consider the LP-relaxation (P) of an integer linear program (ILP) that we obtain by dropping the requirement that variables need to be integral (below). Including fractional solutions allows us to tap into fundamental results from LP duality, which can be used to bound the quality of a solution to (P) by considering the dual (D). Intuitively, the dual LP computes the best lower bound on the objective function value of (P) that we can obtain by a linear combination of its inequalities, where $y_j$ is the weight of the $j$-th inequality in this combination. The constraints of the dual ensure that the contribution of no primal variable to this combination of inequalities exceeds its contribution to the primal objective, i.e., the linear combination lower bounds the objective function. This immediately implies *weak duality*: the fact that every feasible solution to (D) gives a lower bound for the optimum value of (P).

(P)

$$\begin{array}{lll} \min & \sum_{i=1}^{n} c_i x_i & \\ \text{s.t.} & \sum_{i=1}^{n} a_{ij} x_i \geq b_j, & \forall j \in \{1,\dots,m\} \\ & x_i \geq 0, & \forall i \in \{1,\dots,n\} \end{array}$$

(D)

$$\begin{array}{lll} \max & \sum_{j=1}^{m} b_j y_j & \\ \text{s.t.} & \sum_{j=1}^{m} a_{ij} y_j \leq c_i, & \forall i \in \{1,\dots,n\} \\ & y_j \geq 0, & \forall j \in \{1,\dots,m\} \end{array}$$

**Theorem 7.1** (weak duality)**.** *Let $\boldsymbol{x}, \boldsymbol{y}$ be feasible solutions for the dual linear programs (P) and (D). Then,*

$$\sum_{i=1}^{n} c_i x_i \geq \sum_{j=1}^{m} b_j y_j.$$

*Proof.* We have

$$\sum_{i=1}^{n} c_i x_i \overset{\text{(D)}}{\geq} \sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij} y_j x_i = \sum_{j=1}^{m} y_j \sum_{i=1}^{n} a_{ij} x_i \overset{\text{(P)}}{\geq} \sum_{j=1}^{m} b_j y_j.$$

□

In particular, weak duality allows us to prove near-optimality of a solution by providing a feasible dual solution with similar objective function value.

**Corollary 7.2.** *Let $\boldsymbol{x}, \boldsymbol{y}$ be feasible solutions for the dual linear programs (P) and (D) with $\sum_{i=1}^{n} c_i x_i \leq \alpha \sum_{j=1}^{m} b_j y_j$ for some $\alpha \geq 1$. Then $\boldsymbol{x}, \boldsymbol{y}$ are $\alpha$-approximations of the optimum for (P) and (D), respectively.*

It is not clear that we can generally expect to get a good bound on the optimal objective function value only by a linear combination of inequalities. *Strong duality* provides the fundamental guarantee that we can always get a tight bound this way. This very powerful result is at the core of LP duality theory. For its proof we refer to any introductory class on linear optimization.

**Theorem 7.3** (strong duality)**.** *A linear program (P) is feasible and bounded if and only if its dual (D) is feasible and bounded. In this case, the optimum values of (P) and (D) coincide.*

Another fundamental result of duality theory is the principle of *complementary slackness*. This principle states that a pair of feasible solutions of (P) and (D) is optimal if and only if it has the property that a variable of (P)/(D) is non-zero if the corresponding constraint of (D)/(P) is tight and vice-versa. We slightly extend this principle by including approximately optimal solutions with approximately tight constraints for non-zero variables.

**Theorem 7.4** (approximate complementary slackness)**.** *Let $\boldsymbol{x}, \boldsymbol{y}$ be feasible solutions for the dual linear programs (P) and (D), such that the following conditions hold for $\alpha, \beta \geq 1$:*

*(i) primal slackness: If $x_i > 0$ for $i \in \{1,\dots,n\}$, then $\sum_{j=1}^{m} a_{ij} y_j \geq c_i/\alpha$.*

*(ii) dual slackness: If $y_j > 0$ for $j \in \{1,\dots,m\}$, then $\sum_{i=1}^{n} a_{ij} x_i \leq \beta b_j$.*

*Then, $\sum_{i=1}^{n} c_i x_i \leq \alpha\beta \sum_{j=1}^{m} b_j y_j$ and hence $\boldsymbol{x}, \boldsymbol{y}$ are $\alpha\beta$-approximations for (P) and (D), respectively.*

*Proof.* We have

$$\begin{aligned} \sum_{i=1}^{n} c_i x_i &\overset{(i)}{\leq} \sum_{i=1}^{n} \alpha x_i \sum_{j=1}^{m} a_{ij} y_j \\ &= \alpha \sum_{j=1}^{m} y_j \sum_{i=1}^{n} a_{ij} x_i \\ &\overset{(ii)}{\leq} \alpha\beta \sum_{j=1}^{m} b_j y_j. \end{aligned}$$

□

With these structural results in mind, we are ready to state the *primal-dual method*. The idea of *primal-dual approximation algorithms* is to compute not only a primal solution, but to simultaneously compute a feasible dual solution. If we can ensure that the ratio between the value of both solutions is bounded, weak duality (Corollary 7.2) implies a bounded approximation ratio for the primal solution. We use the same method to devise primal-dual online algorithms that maintain a pair of feasible solutions, one for the primal and one for the dual LP, while ensuring that the ratio between the value of both solutions stays bounded.

**Theorem 7.5** (primal-dual method)**.** *Let $\rho \geq 1$ and* ALG *be an online algorithm for the online ILP problem that, in every step $i$, computes feasible solutions $\boldsymbol{x}_i, \boldsymbol{y}_i$ for the LP relaxation of $(A_i, \boldsymbol{b}_i, \boldsymbol{c}_i)$ and its dual $(A_i^\mathsf{T}, \boldsymbol{c}_i, \boldsymbol{b}_i)$, such that $\boldsymbol{x}_i$ is integral and $\boldsymbol{c}_i^\mathsf{T}\boldsymbol{x}_i \leq \rho \boldsymbol{b}_i^\mathsf{T}\boldsymbol{y}_i$. Then* ALG *is strictly $\rho$-competitive.*

In order to illustrate the primal-dual method on a simple example, we apply it to the ski rental problem. To do this, we first need to formulate the LP-relaxation (SRP) of the ski rental ILP (SR) and its dual (SRD).

(SRP)

$$\begin{array}{lll} \min & B \cdot x + \sum_{i=1}^{n} z_i & \\ \text{s.t.} & x + z_j \geq 1, & \forall j \in \{1, \ldots, n\} \\ & x, z_i \geq 0, & \forall i \in \{1, \ldots, n\} \end{array}$$

(SRD)

$$\begin{array}{lll} \max & \sum_{j=1}^{n} y_j & \\ \text{s.t.} & \sum_{j=1}^{n} y_j \leq B & \\ & 0 \leq y_j \leq 1, & \forall j \in \{1, \ldots, n\} \end{array}$$

In every online step of the ILP version of the ski rental problem, we are presented with an additional variable $z_j$ and an additional constraint $x + z_j \geq 1$. We need to satisfy this constraint by either setting $x = 1$ (i.e., by buying skis) or by setting $z_j = 1$ (i.e., by renting skis). At the same time, we want to set $y_j$ in order to maintain a feasible dual solution. The simplest approach is to set $y_j$ as large as possible without violating dual constraints, i.e., such that $\sum_{j=1}^{n} y_j = B$ or $y_j = 1$, whichever happens first when increasing $y_j$ from 0. We can let the dual solution guide our primal decisions by setting $x = 1$ if the corresponding dual constraint $\sum_{j=1}^{n} y_j \leq B$ becomes tight, and setting $z_j = 1$ otherwise (in that case, $y_j = 1$).

Observe that this algorithm is equivalent to the algorithm $\text{ALG}^B$ that always buys on day $B$. We already saw in Section 1.3 that $\text{ALG}^B$ is strictly $(2 - 1/B)$-competitive. We now derive a slightly weaker bound using LP duality theory.

**Theorem 7.6.** $\text{ALG}^B$ *is strictly 2-competitive.*

*Proof.* We use the primal-dual method (Theorem 7.5). By definition, the LP-interpretation of $\text{ALG}^B$ maintains integral feasible solutions of (SRP) and (SRD). It remains to show that $B \cdot x + \sum_{i=1}^{n} z_i \leq 2 \sum_{j=1}^{n} y_j$. We can easily see this using the approximate complementary slackness conditions (Theorem 7.4). For primal slackness, observe that if $x > 0$, then, by integrality of $x$, we have $x = 1$ and thus $\sum_{j=1}^{n} y_j = B$ must hold, by definition of $\text{ALG}^B$. Also, if $z_i > 0$, then $z_i = 1$ and thus we have $y_i = 1$, again by definition of $\text{ALG}^B$. For dual slackness, we simply observe that $x + z_j \leq 2$, i.e., the primal constraints are always tight up to a factor of 2. By Theorem 7.4, we get $B \cdot x + \sum_{i=1}^{n} z_i \leq 2 \sum_{j=1}^{n} y_j$, i.e., the primal solution is a 2-approximation. □

## 7.2 Randomized rounding

The primal-dual method as stated in Theorem 7.5 requires that, in every step $i$, the computed solution $\boldsymbol{x}_i$ of the LP relaxation must be integral. This requirement is very restrictive, and it is often easier to formulate a primal-dual algorithm that gradually increases variables, e.g., the variable $x$ in the ski rental problem. Of course, the fractional solution that is computed this way is not a feasible solution to the original ILP. However, we can obtain a randomized feasible solution by interpreting fractional variable values (smaller 1) as probabilities of setting the corresponding

variable to 1. This way, we can hope to retain the solution quality of the fractional solution by introducing randomization. This approach is the so-called *randomized rounding* technique.

For example, consider again the ski rental problem, this time allowing fractional solutions. On day $i$, the additional primal variable $z_i$ is presented, together with the primal constraint $x + z_i \geq 1$, which corresponds to a new dual variable $y_i$. We define a natural primal-dual algorithm PD that computes a fractional solution in every step. First, if $x < 1$, we set $y_i = 1$ and otherwise we set $y_i = 0$. Then, in order to satisfy $x + z_i \geq 1$, we set $z_i = 1 - x$. Finally, PD increases $x$ by some amount $\alpha(x + \beta)$ to a maximum of 1, where $\alpha, \beta \in \mathbb{R}$ will be determined later. Since $x$ grows monotonically over time, this ensures that the primal constraints always stay satisfied.

We show that PD computes a good fractional solution using the primal-dual method (without requiring integrality). To do this, we need to show that the primal and dual solutions computed in every step are feasible and that the primal objective function value remains bounded by $\rho \geq 1$ times the dual objective function value (analogously to Theorem 7.5).

We already argued that the primal solution is feasible, because the constraints $x + z_i \geq 1$ remain satisfied by definition of PD. To show dual feasibility, we need to ensure $\sum_{j=1}^{n} y_j \leq B$, i.e., $x \geq 1$ after at most $B$ days. Let $x_i \geq 0$ denote the increment of $x$ in step $i$, i.e., $x = \sum_{i=1}^{n} x_i$ at the end, $x_1 = \alpha\beta$, and

$$x_i := \alpha(\sum_{j=1}^{i-1} x_j + \beta) = \alpha x_{i-1} + \alpha(\sum_{j=1}^{i-2} x_j + \beta) = (1+\alpha)x_{i-1} = (1+\alpha)^{i-1}\alpha\beta,$$

for $i \in \{2, \ldots, n\}$. After $B$ days we have

$$x = \sum_{j=1}^{B} x_j = \sum_{j=0}^{B-1} (1+\alpha)^j \alpha\beta = \alpha\beta \frac{(1+\alpha)^B - 1}{(1+\alpha) - 1} = \beta \cdot [(1+\alpha)^B - 1].$$

To ensure that $x = 1$ after $B$ days, and have dual feasibility, we may thus set

$$\beta := 1/[(1+\alpha)^B - 1].$$

In order to bound the ratio between the total primal and dual objective function value in the end, we can bound the ratio between the change in primal and dual objective function values in every step. Consider step $i$. We may assume $x = x_{i-1} < 1$ at the start of step $i$, otherwise the objective function values do not change, since $z_i, y_i$ are set to 0 and $x$ remains 1. But then $y_i$ is set to 1, which increases the dual objective by 1. On the other hand, the primal objective changes by

$$Bx_i + z_i = B\alpha(\sum_{j=1}^{i-1} x_j + \beta) + 1 - \sum_{j=1}^{i-1} x_j.$$

We can set $\alpha = 1/B$ in order to eliminate the dependence on $x_{i-1}$. We get a constant change in primal objective of exactly $1 + \beta$ times the change in dual objective in every step. Consequently, the ratio between primal and dual objective is always bounded by $1 + \beta$. Using weak duality, we get the following result.

**Theorem 7.7.** PD *computes a* $(1+\beta)$*-approximate, fractional solution for the ski rental problem, with* $\beta = 1/[(1+1/B)^B - 1]$.

Based on the fractional solution computed by PD, we can define a randomized algorithm RPD using randomized rounding. To do this, we interpret the fractional contribution $x_i$ to $x$ in step $i$ as the probability of buying skis on day $i$ and the final value of $x$ as the probability of buying skis at all. We can achieve these probabilities by initially choosing a threshold $\tau \in [0, 1]$ uniformly at random and buying skis as soon as $x$ exceeds $\tau$. The expected cost for buying skis then becomes

$B \cdot x = B \cdot \sum_{i=1}^{n} x_i$. The probability of renting skis on the $i$-th day is $1 - \sum_{j=1}^{i-1} x_j = z_i$, thus the total expected cost of the rounded solution is

$$\mathbb{E}_{\tau \sim [0,1]}[\textsc{Rpd}] = Bx + \sum_{i=1}^{n} z_i,$$

which coincides with the cost of the fractional solution produced by Pd.

**Theorem 7.8.** Rpd *is a strictly* $(1+\beta)$*-competitive randomized online algorithm for ski rental, with* $\beta = 1/[(1+1/B)^B - 1]$.

*Remark* 7.9. For $B \to \infty$, we have $\beta = \frac{1}{e-1}$ and the resulting randomized competitive ratio of Rpd of $\frac{e}{e-1}$ is best possible for the ski rental problem.

## 7.3 Online bipartite matching

An important example of a primal-dual online algorithm is the Ranking algorithm for the online bipartite matching problem. In this problem, the set $L$ of vertices (the "left-hand side") of a bipartite graph $G = (L \cup R, E)$ is given and the vertices of $R$ arrive online, together with their incident edges to vertices in $L$. Our goal is to match each vertex of $R$ to at most one unmatched neighbor in $L$ as it arrives, such as to maximize the size of the final matching. Formally, we define matchings as follows.

**Definition 7.10.** A *matching* in a graph $G = (V, E)$ is a set $M \subseteq E$ of mutually disjoint edges. A *maximum matching* is a matching of maximum cardinality, and a *maximal matching* is a matching that is no proper subset of any other matching.

| Online Bipartite Matching Problem | |
|---|---|
| **given:** | set of left-hand vertices $L$ of an unknown bipartite graph $G = (L \cup R, E)$ |
| **online:** | sequence $\sigma = (v_1, v_2, \ldots, v_n) \in R$ of right-hand vertices, each $v_i$ given by the set $L_i \subseteq L$ of neighbors of $v_i$ in $G$ |
| **actions** (step $i$)**:** | match $v_i$ to an unmatched vertex in $L_i$, or leave it unmatched |
| **objective:** | maximize the size of the matching |

Note that the online bipartite matching problem is a *maximization* problem. Accordingly, we need to adapt the definition of competitiveness. The definition of randomized competitiveness and the (randomized) competitive ratio are changed analogously.

**Definition 7.11.** A deterministic online algorithm Alg is $\rho$*-competitive* for a maximization problem if there is a constant $\alpha \geq 0$, such that for all instances $\sigma \in \Sigma$

$$\textsc{Alg}(\sigma) \geq \frac{1}{\rho} \cdot \textsc{Opt}(\sigma) - \alpha.$$

The statement of the primal-dual method needs to be adapted as well.

**Theorem 7.12** (primal-dual method (maximization))**.** *Let* $\rho \geq 1$ *and* Alg *be an online algorithm for maximization version of the online ILP problem that, in every step* $i$*, computes feasible solutions* $\boldsymbol{x}_i, \boldsymbol{y}_i$ *for the LP relaxation of* $(A_i, \boldsymbol{b}_i, \boldsymbol{c}_i)$ *and its dual* $(A_i^\mathsf{T}, \boldsymbol{c}_i, \boldsymbol{b}_i)$*, such that* $\boldsymbol{x}_i$ *is integral and* $\boldsymbol{c}_i^\mathsf{T} \boldsymbol{x}_i \geq (1/\rho) \boldsymbol{b}_i^\mathsf{T} \boldsymbol{y}_i$*. Then* Alg *is strictly* $\rho$*-competitive.*

### 7.3.1 Deterministic algorithms

We first give a simple lower bound on the best deterministically possible competitive ratio.

**Theorem 7.13.** *Every deterministic online matching algorithm has a competitive ratio of at least* 2.

*Proof.* Consider an adversarial request sequence $\sigma = (v_1, v_2)$ with two requests for the online bipartite matching problem, where $v_1$ has an edge to two vertices $u_1, u_2 \in L$ and $v_2$ has a single edge to the vertex in $\{u_1, u_2\}$ that was matched to $v_1$ in the first step. In this instance, we obviously have $\textsc{Opt}(\sigma) = 2$ and $\textsc{Alg}(\sigma) = 1$ for any online algorithm Alg. We can arbitrarily extend this example to a request sequence $\sigma = (v_1, v_2, \dots, v_{2n})$ and a set $L = \{u_1, \dots, u_{2n}\}$ for any $n \in \mathbb{N}$, by, for every $i \in \{1, \dots, n\}$ letting $v_{2i-1}$ have an edge to both $u_{2i-1}, u_{2i}$, and letting $v_{2i}$ have an edge to the single matched vertex from $\{u_{2i-1}, u_{2i}\}$. Then $\textsc{Opt}(\sigma) = 2n$, while $\textsc{Alg}(\sigma) \leq n$, and thus $\lim_{n\to\infty} \frac{\textsc{Opt}(\sigma)}{\textsc{Alg}(\sigma)-\alpha} \geq 2$ for any $\alpha \in \mathbb{R}$. Note that our construction works for any algorithm Alg that deterministically decides whether to match $v_{2i-1}$ to $u_{2i-1}$ or $u_{2i}$ for every $i \in \{1, \dots, n\}$. We thus get a lower bound of 2 on the competitive ratio of any deterministic online algorithm, analogously to Proposition 1.12. □

We can achieve this competitive ratio easily, using the straight forward Greedy algorithm that starts by ordering $L$ arbitrarily and then matches every incoming right-hand vertex to the first unmatched neighbor in this order.

**Theorem 7.14.** Greedy *is strictly* 2*-competitive for the online bipartite matching problem.*

*Proof.* Observe that Greedy always computes a maximal matching $M_{\textsc{Greedy}}$: We need to show that no edge $e = \{u, v\} \in E \setminus M_{\textsc{Greedy}}$ can be added to $M_{\textsc{Greedy}}$ to obtain a larger matching. Either $v \in R$ is matched by Greedy or $u \in L$ was already matched when $v$ appeared, otherwise Greedy would have matched $u$ and $v$ when $v$ appeared. Thus $M_{\textsc{Greedy}} \cup \{e\}$ is not a matching.

It is now sufficient to show that *every* maximal matching $M$ has at least half the size of the maximum matching $\textsc{Opt}[\sigma]$. To see this, we compare the number of matched vertices in $M$ and $\textsc{Opt}[\sigma]$. Every matching edge $e \in \textsc{Opt}[\sigma]$ is responsible for two matched vertices in $\textsc{Opt}[\sigma]$. On the other hand, since $M$ is maximal, at least one endpoint of $e$ is matched in $M$. Overall, the number of matched vertices is $2\,|\textsc{Opt}(\sigma)|$ in $\textsc{Opt}[\sigma]$ versus at least $|\textsc{Opt}(\sigma)|$ in $M$. □

### 7.3.2 Ranking algorithm

We can avoid the simple lower bound construction of Theorem 7.13 by making the Greedy algorithm less predictable. The easiest way to do this is to start with a random order of $L$ and then to deterministically execute Greedy for this order. The Ranking algorithm is alternate formulation of this exact algorithm: For every $u \in L$ pick a random number $g_u \in [0, 1]$ uniformly at random, and then, in every step $i$, match $v_i$ to the unmatched neighbor $u \in L$ with the lowest value $g_u$. We will interpret $g_u$ as the *greed* of $u$, which means that Ranking matches $v_i$ to its least greedy (i.e., most "generous") unmatched neighbor. [4]

We analyze Ranking using the primal-dual method. To do this, we first need to formulate the primal and dual linear programs. Let $x_{uv}$ be a variable that indicates whether the edge $\{u, v\} \in E$ is part of the matching ($x_{uv} = 1$), or not ($x_{uv} = 0$). By allowing fractional values of $x_{uv}$ we get the linear program relaxation (BMP) with its dual (BMD):

(BMP)

| | | |
|---|---|---|
| max | $\sum_{\{u,v\}\in E} x_{uv}$ | |
| s.t. | $\sum_{v:\{u,v\}\in E} x_{uv} \leq 1$, | $\forall u \in L$ |
| | $\sum_{u:\{u,v\}\in E} x_{uv} \leq 1$ | $\forall v \in R$ |
| | $x_{uv} \geq 0$, | $\forall \{u, v\} \in E$ |

(BMD)

| | | |
|---|---|---|
| min | $\sum_{u\in L} \alpha_u + \sum_{v\in R} \beta_v$ | |
| s.t. | $\alpha_u + \beta_v \geq 1$ | $\forall \{u, v\} \in E$ |
| | $\alpha_u \geq 0$ | $\forall u \in L$ |
| | $\beta_v \geq 0$ | $\forall v \in R$ |

In order to turn Ranking into a primal-dual algorithm, we additionally need to maintain a dual solution in every step. Whenever Ranking matches two vertices $u \in L, v \in R$, the primal objective

[4] Note we may safely ignore the event that a vertex has greed 1 or that two vertices have the same greed, since the combined probability of any such event occurring is 0.

increases by $x_{uv} = 1$. The primal-dual method for maximization problems (Theorem 7.12) requires that, at the same time, the dual increases by at most $\rho$, where $\rho \geq 1$ is the competitive ratio we want to prove. We can achieve this by distributing a value of $\rho$ between $\alpha_u$ and $\beta_v$. We do this according to the greed of $u$, by setting

$$\alpha_u = \rho \cdot h(g_u) \cdot \sum_{\{u,v\} \in E} x_{uv}, \quad \beta_v = \rho \cdot \sum_{\{u,v\} \in E} (1 - h(g_u)) \cdot x_{uv},$$

where $h\colon [0,1] \to [0,1]$ is some monotonically increasing function with $h(1) = 1$ that we will determine later. Note that as long as $u \in L$ (respectively, $v \in R$) are unmatched, we have $\alpha_u = 0$ (respectively, $\beta_v = 0$).

We interpret the dual variables $\alpha, \beta$ as some distribution of the dual objective value over the vertices in $L, R$, respectively. Each vertex $u \in L$ that is being matched gets a value of $\rho$ that it distributes between itself and its matching partner according to its greed $g_u$: It keeps $\rho \cdot h(g_u)$ to itself and gives $\rho \cdot (1 - h(g_u))$ to its partner. Whenever a new vertex $v \in R$ appears, it chooses the most generous available neighbor $u \in L$ as its matching partner in order to get the highest possible value.

Fix any edge $\{u, v\} \in E$ with $u \in L$ and $v \in R$ and assume we run Ranking, ignoring the vertex $u \in L$ and using the same greed values $g_{u'}$ for all $u' \in L \setminus \{u\}$. We denote this modified algorithm by $\textsc{Ranking}_{\bar{u}}$. We are interested in the dual utility value $\bar{\beta}_{uv}$ that $v$ gets during the execution of $\textsc{Ranking}_{\bar{u}}$. Equivalently, we can ask for the greed value $\bar{g}_{uv}$ of $v$'s matching partner (if $v$ gets matched) in the execution of $\textsc{Ranking}_{\bar{u}}$. Formally, we define

$$\bar{g}_{uv} = \begin{cases} g_{u'} & \text{if } v \text{ gets matched to } u' \in L \text{ by } \textsc{Ranking}_{\bar{u}}, \\ 1 & \text{if } v \text{ remains unmatched.} \end{cases}$$

and $\bar{\beta}_{uv} := \rho \cdot (1 - h(\bar{g}_{uv}))$.

**Lemma 7.15.** *If $g_u < \bar{g}_{uv}$ for some edge $\{u, v\} \in E$ with $u \in L$, then $u$ gets matched by* Ranking.

**Lemma 7.16.** *For all $\{u, v\} \in E$ with $u \in L$ and $v \in R$, we have $\beta_v \geq \bar{\beta}_{uv}$.*

By definition of Ranking, we always compute a feasible and integral primal solution (a matching). Since every increase in primal objective increases the dual objective by exactly $\rho$, we maintain a ratio of $\rho$ between primal and dual objective. The final ingredient of the primal-dual method is to use dual feasibility and apply weak duality in order to obtain a guarantee on the approximation quality of the computed primal solution. Unfortunately, we cannot guarantee the computed dual solution to always be feasible!

Recall that for a bounded randomized competitive ratio we only need a guarantee on the *expected* approximation quality of computed solution and not for every possible outcome. Since the ratio between primal and dual objective is $\rho$ in every execution of Ranking, the same is true for the ratio between the expected primal and dual objective. By linearity of expectation, we get

$$\rho = \frac{\mathbb{E}\left[\sum_{u \in L} \alpha_u + \sum_{v \in R} \beta_v\right]}{\mathbb{E}\left[\sum_{\{u,v\} \in E} x_{uv}\right]} = \frac{\sum_{u \in L} \mathbb{E}[\alpha_u] + \sum_{v \in R} \mathbb{E}[\beta_v]}{\sum_{\{u,v\} \in E} \mathbb{E}[x_{uv}]}.$$

We can therefore apply the primal-dual method using the expected primal solution $\mathbb{E}[\boldsymbol{x}]$ and the expected dual solution $\mathbb{E}[\boldsymbol{\alpha}], \mathbb{E}[\boldsymbol{\beta}]$. Since the feasible region of the primal LP is a convex polytope, the expected solution $\mathbb{E}[\boldsymbol{x}]$ is feasible. The expected primal solution $\mathbb{E}[\boldsymbol{x}]$ may be fractional, but it is only important that every possible computed solution is integral. In order to complete the primal-dual argument we need to apply weak duality. To do this, it remains to show that the expected dual solution is feasible, i.e., that $\mathbb{E}[\alpha_u] + \mathbb{E}[\beta_v] \geq 1$ for all $\{u, v\} \in E$ with $u \in L$ and $v \in R$. It then follows that the optimal fractional solution is bracketed between the primal and dual objective values, which are separated by a ratio of $\rho$.

To bound $\mathbb{E}[\alpha_u]$, we fix the values $g_{u'}$ for every $u' \in L \setminus \{u\}$ to any values in $[0,1]$, and only consider the expectation with respect to choosing $g_u$ uniformly at random from $[0,1]$. Recall that if $u$ is matched by RANKING, then $\alpha_u = \rho \cdot h(g_u)$, otherwise $\alpha_u = 0$. By Lemma 7.15, we know that $u$ is guaranteed to be matched when $g_u < \bar{g}_{uv}$. We also know that $\bar{g}_{uv}$ and thus $\bar{\beta}_{uv}$ are independent of $g_u$. We thus have

$$\mathbb{E}_{g_u \sim [0,1]}[\alpha_u] \geq \rho \cdot \int_0^{\bar{g}_{uv}} h(g)\,\mathrm{d}g =: \rho[H(\bar{g}_{uv}) - H(0)],$$

where $H$ is the antiderivative of $h$. On the other hand, by Lemma 7.16, we have

$$\mathbb{E}_{g_u \sim [0,1]}[\beta_v] \geq \bar{\beta}_{uv} = \rho \cdot (1 - h(\bar{g}_{uv})).$$

Together, the expected dual solution is feasible if, for every choice of $g_{u'}$ for every $u' \in L \setminus \{u\}$, we have

$$H(\bar{g}_{uv}) - H(0) + 1 - h(\bar{g}_{uv}) \geq 1/\rho. \tag{7.1}$$

In order to get rid of the dependency on $\bar{g}_{uv}$, we choose $h\colon [0,1] \to [0,1]$ to be of the form $h(g) = ce^g$. The condition $h(1) = 1$ then immediately implies $c = 1/e$. With this, the smallest possible competitive ratio that guarantees dual feasibility (eq. (7.1)) is $\rho = \frac{e}{e-1}$.

**Theorem 7.17** (Karp et al. [1990])**.** RANKING *is strictly* $\frac{e}{e-1}$*-competitive for online bipartite matching.*

### 7.3.3 Randomized Lower Bound

We now derive a tight lower bound on the randomized competitive ratio of the online bipartite matching problem using Yao's principle for maximization problems:

**Theorem 7.18** (Yao [1977], maximization)**.** *Let* $(q_n\colon \Sigma \to [0,1])_{n\in\mathbb{N}}$ *be a sequence of probability distributions with*

$$\lim_{n\to\infty} \mathbb{E}_{\sigma\sim q_n}[\textsc{Opt}(\sigma)] = \infty.$$

*Then, every randomized online algorithm for a maximization problem has a competitive ratio of at least*

$$\limsup_{n\to\infty} \frac{\mathbb{E}_{\sigma\sim q_n}[\textsc{Opt}(\sigma)]}{\sup_{\textsc{Alg}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_n}[\textsc{Alg}(\sigma)]}.$$

With this, we show the following bound.

**Theorem 7.19** (Karp et al. [1990])**.** *The competitive ratio of any randomized online bipartite matching algorithm is at least* $\frac{e}{e-1}$*.*

*Proof.* To make use of Theorem 7.18, we need to define a sequence $(q_n)_{n\in\mathbb{N}}$ of probability distributions over input sequences. We define $q_n\colon \Sigma \to [0,1]$ to be a distribution over request sequences of length $n$ that results from the following process. We use a set $L$ containing $n$ vertices, and say that all vertices of $L$ are *active* initially. In every step $i$, the appearing vertex $v_i \in R$ has an edge to every active vertex in $L$, and at the end of step $i$ a random active vertex becomes inactive. Equivalently, we set $L = \{u_1, \ldots, u_n\}$ and initially choose a random ordering $u_{j_1}, u_{j_2}, \ldots, u_{j_n}$ of the vertices in $L$. Then $v_i$ is adjacent to the set of vertices $\{u_{j_i}, u_{j_{i+1}}, \ldots, u_{j_n}\}$ of size $n - i + 1$.

By definition, it is possible to match $v_i$ to $u_{j_i}$ in every step $i$, thus $\textsc{Opt}(\sigma) = n$ for every possible input sequence $\sigma$ produced by this process. In particular, we have $\mathbb{E}_{\sigma\sim q_n}[\textsc{Opt}(\sigma)] = n$ and thus $\lim_{n\to\infty} \mathbb{E}_{\sigma\sim q_n}[\textsc{Opt}(\sigma)] = \infty$, as required by Yao's principle.

Now consider the options for an online algorithm in step $i$. If $v_i$ still has unmatched active neighbors, it is never a mistake to match $v_i$ to one of them. By symmetry of the process, all unmatched active neighbors are completely identical and there is no reason to choose one of them over the others.

Therefore, no online algorithm has a better choice than picking a matching partner for $v_i$ uniformly at random from amongst its unmatched active neighbors. We obtain that the corresponding algorithm RAND is a best-possible online algorithm for the specific distribution $q_n$ of instances that we constructed.

Using Yao's principle, we obtain that no online algorithm has competitive ratio below

$$\limsup_{n\to\infty} \frac{\mathbb{E}_{\sigma\sim q_n}[\textsc{Opt}(\sigma)]}{\sup_{\textsc{Alg}\in\mathcal{A}_{\text{det}}} \mathbb{E}_{\sigma\sim q_n}[\textsc{Alg}(\sigma)]} \geq \limsup_{n\to\infty} \frac{n}{\mathbb{E}_{\sigma\sim q_n}[\textsc{Rand}(\sigma)]}. \tag{7.2}$$

It remains to bound $\lim_{n\to\infty} \mathbb{E}_{\sigma\sim q_n}[\textsc{Rand}(\sigma)]$. Let $a_i = n - i + 1$ denote the number of active vertices in $L$ at the start of step $i$, and let $x_i \leq a_i$ denote the number of unmatched active vertices in $L$ at the start of step $i$. Let further $\Delta a_i := a_i - a_{i+1}$ and $\Delta x_i := x_i - x_{i+1}$ denote the change in $a_i$ and $x_i$ during step $i$. We have $\Delta a_i = 1$ by definition of the random process. While $x > 0$, if $u_{j_i}$ is unmatched at the start of step $i$ and $v_i$ does not get matched to $u_{j_i}$, then $\Delta x_i = 2$, and in all other cases $\Delta x_i = 1$. The probability that $u_{j_i}$ is still unmatched, i.e., that $u_{j_i}$ is one of the $x_i$ unmatched active vertices, is $\frac{x_i}{a_i}$, since, by symmetry, every active vertex has the same probability of being matched. The probability that $u_{j_i}$ is not matched to $v_i$, given that it is one of the $x_i$ unmatched active vertices, is $\frac{x_i-1}{x_i}$. We have

$$\mathbb{E}_{\sigma\sim q_n}[\Delta x_i] = 1 + \frac{x_i}{a_i}\frac{x_i - 1}{x_i} = 1 + \frac{x_i - 1}{a_i},$$

and, since $\Delta a_i = 1$,

$$\frac{\mathbb{E}_{\sigma\sim q_n}[\Delta x_i]}{\mathbb{E}_{\sigma\sim q_n}[\Delta a_i]} = 1 + \frac{x_i - 1}{a_i}.$$

For $n \to \infty$ we can approximate the solution to this difference equation, i.e., the distribution over $x_i$'s and $a_i$'s that satisfies it, with high probability[5] by the solution to the ordinary differential equation

$$\frac{\mathrm{d}x}{\mathrm{d}a} = 1 + \frac{x - 1}{a}.$$

The only solution to this differential equation is $x = ca + a\ln a + 1$ for any $c \in \mathbb{R}$. In order to determine $c$, we observe that $x_1 = a_1 = n$, thus

$$c = 1 - \ln n - \frac{1}{n} = \frac{n-1}{n} - \ln n.$$

We get that with high probability

$$x = 1 + a(\frac{n-1}{n} + \ln\frac{a}{n}).$$

Consider the step $i$ where $x_i = 1$, i.e., one vertex in $L$ remains active and unmatched. At this point, in the next step this vertex will still be matched, but then all unmatched vertices in $L$ are inactive and the matching does not grow further. For $x = 1$ we get

$$\ln\frac{a}{n} = -\frac{n-1}{n}$$

or

$$a = ne^{-\frac{n-1}{n}}.$$

For $n \to \infty$, once $x = 1$, the expected number of remaining active vertices in $L$ gets arbitrarily close to $a = n/e$. Since only one of them will still be matched, the total size of the matching computed by RAND is given by

$$\mathbb{E}_{\sigma\sim q_n}[\textsc{Rand}(\sigma)] = n - \frac{n}{e} + 1 = \frac{e-1}{e}n + 1.$$

With eq. (7.2) we obtain the claimed lower bound. □

[5] With $x, a$ approximating $x_i, a_i$ with "high probability", we mean that, for every $\varepsilon > 0$, the probability that the function $x(a)$ has error at most $\varepsilon$ goes to 1 as $n$ goes to infinity.

# 8 Online Load Balancing

Scheduling problems have an inherent time component that makes them particularly well-suited to be studied in an online setting. In scheduling, we are given *jobs* that need to be processed by a set of machines. Each machine can run some jobs better (i.e., faster) than others, which is usually expressed in terms of machine-dependent *processing times* for each job. Scheduling problems have been studied in a multitude of variants, e.g., differing in

- whether each job needs to be executed en bloc on one machine, or can be interrupted (*preempted*) and resumed on the same or a different machine (*migration*), or even can be parallelized;
- whether jobs have a *release time* when they become available, and/or a *deadline* when they need to be completed;
- whether jobs have *precedence constraints* that demand a partial order in which jobs need to be processed;
- whether the latest completion time (*makespan*), the *(weighted) sum of completion times*, or other measures need to be minimized.

We focus on the *load balancing* problem, where each job needs to be processed without interruption on a single machine and we want to minimize the makespan, i.e., the maximum *load* of a machine. In the online variant, we assume that jobs arrive one after the other at time 0 and need to immediately be assigned to a machine. Importantly, jobs *do not* arrive over time, but rather before the actual processing of the computed schedule begins.

We now formally specify the online load balancing problem. Note that it is customary to identify machines with natural numbers $\{1, \dots, m\}$ and jobs with $\{1, \dots, n\}$ to allow for simpler notation, even though each job is formally given by a vector of its processing times on every machine.

| Online Load Balancing Problem | |
|---|---|
| **given:** | machines $1, \dots, m$ |
| **online:** | sequence $\sigma = (1, \dots, n)$ of jobs, each job $j$ comes with processing times $p_{ij} \in \mathbb{R}$ on every machine $i \in \{1, \dots, m\}$ |
| **actions** (for job $j$)**:** | assign job $j$ to some machine $i_j$; we set $J_i(\sigma) := \{j \in \sigma \colon i_j = i\}$ |
| **objective:** | minimize the makespan $\max_{i=\{1,\dots,m\}} L_i(\sigma)$, where $L_i(\sigma) := \sum_{j \in J_i(\sigma)} p_{ij}$ is the load of machine $i$ |

In terms of this notation, $C_j(\sigma) := L_{i_j}(\sigma_{\leq j})$ is the completion time of job $j$, and $S_j(\sigma) := L_{i_j}(\sigma_{\leq j-1})$ is its starting time. See Figure 10 for an example.

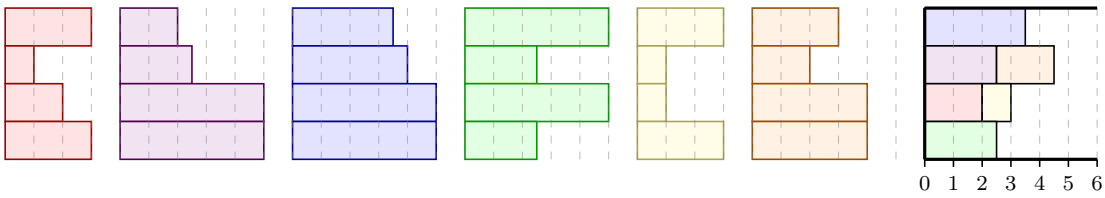


Figure 10: Example of a load balancing instance with $m = 4$ machines and $n = 6$ jobs. The unique optimum solution with maximum load 4.5 is depicted on the right.

## 8.1 Identical machines

We first consider online load balancing with *identical machines*, where we demand that the processing time of a job does not depend on the machine the job is processed on, i.e., $p_{ij} = p_j$. A very natural online algorithm for this setting is *Graham's list scheduling algorithm* (List): Schedule each job $j$ on a machine $i$ that currently has minimum load $L_i(\sigma_{\leq j-1})$. This algorithm was proposed by Graham [1969] and is often considered to be the first approximation algorithm, though it predates the theory of NP-hardness.

**Theorem 8.1** (Graham [1969])**.** List *has strict competitive ratio* $2 - 1/m$ *for the online load balancing problem with identical machines.*

*Proof.* Fix an instance given by sequence $\sigma = (1, \ldots, n)$ and associated processing times $p_j \in \mathbb{R}$ for $j \in \sigma$. Since Opt needs to schedule every job, we trivially have $\textsc{Opt}(\sigma) \geq \max_{j\in\sigma} p_j$. We get another simple lower bound by observing that the total load $L = \sum_{i=1}^{m} L_i(\sigma) = \sum_{i=1}^{m} \sum_{j\in J_i} p_j = \sum_{j\in\sigma} p_j$ is the same for any schedule and, therefore,

$$\textsc{Opt}(\sigma) \geq L/m = \sum_{j\in\sigma} p_j/m.$$

Now take any job $j \in \sigma$ that has maximum completion time $C_j = \max_{j'\in\sigma} C_{j'}$. By definition of List, machine $i_j$ had minimum load at the starting time $S_j$ of job $j$, i.e., all machines are busy until at least $S_j$, i.e., $\frac{1}{m}\sum_{j'=1}^{j-1} p_{j'} \geq S_j$. Hence,

$$\begin{aligned}
\textsc{List}(\sigma) &= C_j \\
&= S_j + p_j \\
&\leq \sum_{j'\in\sigma\setminus\{j\}} p_{j'}/m + p_j \\
&= \sum_{j'\in\sigma} p_{j'}/m + (1 - 1/m)p_j \\
&\leq \sum_{j'\in\sigma} p_{j'}/m + (1 - 1/m) \max_{j'\in\sigma} p_{j'} \\
&\leq (2 - 1/m) \cdot \textsc{Opt}(\sigma).
\end{aligned}$$

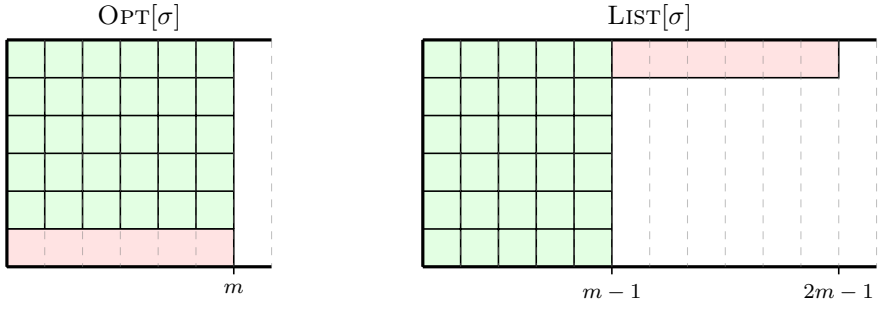


Figure 11: Lower bound construction for List with $m(m-1)$ jobs with unit processing times followed by one job with processing time $m$.

It remains to show a tight lower bound on the strict competitive ratio of List. Consider a sequence $\sigma = (1, \ldots, n)$ with $n = m(m-1)+1$ and $p_{j\neq n} = 1, p_n = m$ (see Figure 11). Since List attempts to keep the load on all machines balanced, until step $n-1$ it will produce load $L_i(\sigma_{\leq n-1}) = m-1$ on all machines $i$. It immediately follows that $\textsc{List}(\sigma) = 2m-1$. On the other hand, we can achieve the same load on all machines by assigning job $n$ to its own machine and distributing all other jobs

evenly. It follows that $\textsc{Opt}(\sigma) = m$, and thus $\textsc{List}(\sigma) \geq (2 - 1/m)\textsc{Opt}(\sigma)$, which establishes a matching lower bound. □

**Proposition 8.2.** List *is a best-possible online algorithm for* $m = 2$.

*Proof.* Consider the instance $\sigma$ with processing times $1, 1, 2$ and let Alg be an online algorithm with best-possible competitive ratio. Since any online algorithm that schedules the two first jobs on the same machine only has competitive ratio $2 > 2 - 1/m = 3/2$ for the instance $\sigma_{\leq 2}$, we can conclude that Alg schedules the first two jobs on different machines. But then, Alg has makespan 3 for instance $\sigma$, while $\textsc{Opt}(\sigma) = 2$. We get $\textsc{Alg}(\sigma) \geq 3\textsc{Opt}(\sigma)/2 = (2 - 1/m) \cdot \textsc{Opt}(\sigma)$. □

*Remark* 8.3. The best known algorithm for an arbitrary number of identical machines was presented by Fleischer and Wahl [2000] and is 1.9201-competitive. The best known lower bound of 1.88 was given by Rudin III and Chandrasekaran [2003].

## 8.2 Related machines

Load balancing with *related machines* allows different processing times on each machine by assigning each machine $i$ a speed $s_i$, $s_1 \leq s_2 \leq \cdots \leq s_m$, while demanding $p_{ij} = p_j/s_i$ for all jobs $j$, where $p_j > 0$. This allows machines to be different as long as this difference (in speed) remains consistent between jobs. If we want to apply List to this setting, it makes a difference whether we choose a machine with minimum load *before* scheduling the next job on this machine (preGreedy) or *after* scheduling the next job on it (postGreedy). If we have two machines with very different speeds, preGreedy is arbitrarily bad (proportional to the ratio between the speeds) already for two identical jobs, since the first job may be put on the slow machine. On the other hand, postGreedy has a logarithmically bounded competitive ratio (without proof).

**Theorem 8.4** (upper bound: Cho and Sahni [1988], lower bound: Aspnes et al. [1997])**.** postGreedy *has competitive ratio* $\Theta(\log n)$ *for the online load balancing problem on related machines.*

We develop a better online algorithm that uses a typical design scheme: Assume that we know an upper bound $\Lambda \geq \textsc{Opt}(\sigma)$ and are able to design a procedure that, given $\Lambda$, produces a solution of cost below $\rho \cdot \Lambda$ for some constant $\rho$. Now make a guess $\Lambda$ for the value of $\textsc{Opt}(\sigma)$ and try to use our procedure. We either get a solution of cost below $\rho\Lambda$, or we realize that our guess was incorrect. We can then begin with $\Lambda = \Lambda_0 \leq \textsc{Opt}(\sigma)$ and restart from the beginning with $\Lambda_i = 2\Lambda_{i-1}$ every time our procedure fails. For every restart, we ignore all choices made so far (but not their costs), and pretend that the input sequence arrives again online from the beginning (even though we know part of the sequence already).

Since $\Lambda_0 > 0$, the procedure eventually succeeds for $\Lambda_k$, $k \in \mathbb{N}$ (at the latest when $\Lambda_k \geq \textsc{Opt}(\sigma)$). If $k = 0$, we computed a solution of cost at most $\rho\Lambda_0 \leq \rho \cdot \textsc{Opt}(\sigma)$. Otherwise, we know that $\Lambda_{k-1} < \textsc{Opt}(\sigma)$. Even summing up the costs for every run independently (see Figure 12), i.e., scheduling the same job multiple times, we get $\textsc{Alg}(\sigma) \leq \sum_{i=0}^{k} \rho\Lambda_i = \rho \sum_{i=0}^{k} 2^i\Lambda_0 = \rho\frac{2^{k+1}-1}{2-1}\Lambda_0 < \rho 2^{k+1}\Lambda_0$. Using $2^{k-1}\Lambda_0 = \Lambda_{k-1} < \textsc{Opt}(\sigma)$ we get $\textsc{Alg}(\sigma) < 4\rho \cdot \textsc{Opt}(\sigma)$.

**Lemma 8.5** (doubling strategy)**.** *If there is an online algorithm that computes a solution of cost at most* $\rho\Lambda$ *for given* $\Lambda \geq \textsc{Opt}(\sigma)$ *and we can compute some bound* $0 < \Lambda_0 \leq \textsc{Opt}(\sigma)$ *in the first step, then we can devise a strictly* $4\rho$*-competitive online algorithm.*

We introduce the algorithm SlowFit that uses the doubling strategy (for $\rho = 2$) with the following subroutine for a given bound $\Lambda \geq \textsc{Opt}(\sigma)$: Put every job $j$ on the slowest machine $i$ where the resulting load does not exceed $2\Lambda$ (the *slowest* machine where the job *fits*), i.e., $i = \min\{i' \in \{1, \ldots, m\} \colon L_{i'}(\sigma_{\leq j-1}) + \frac{p_j}{s_{i'}} \leq 2\Lambda\}$. The following lemma guarantees that if $\Lambda \geq \textsc{Opt}(\sigma)$, such a machine must always exist. If this is not the case, then SlowFit can safely abort the subroutine and restart it for $\Lambda' = 2\Lambda$.

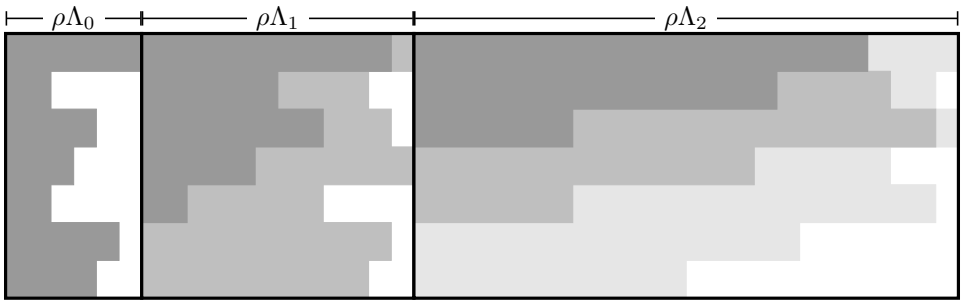


Figure 12: Illustration of the doubling strategy for scheduling. If our subroutine is unable to fit all jobs into a time window of width $\rho\Lambda_i$, it restarts from the beginning with a wider window of width $2\Lambda_{i+1}$. The shading of jobs corresponds to the smallest time window in which they could be scheduled. Observe that jobs are generally scheduled in multiple time windows and thus contribute multiple times to the overall cost.

**Lemma 8.6.** *Given* $\Lambda \geq \textsc{Opt}(\sigma)$ *the subroutine of* SlowFit *computes a solution of cost at most* $2\Lambda$.

*Proof.* We need to show that the subroutine of SlowFit always finds a machine where to schedule a job $j$ without exceeding a load of $2\Lambda$. For the sake of contradiction, assume this is not the case, i.e., there is a job $j$ for which $L_i(\sigma_{\leq j-1}) + \frac{p_j}{s_i} > 2\Lambda$ for all machines $i \in \{1, \ldots, m\}$. Let $L_i^\star(\sigma)$ denote Opt's load on machine $i$ after processing the entire request sequence $\sigma$. Since $\textsc{Opt}(\sigma) \leq \Lambda$ and $p_j > 0$, we know that $\sum_{i=1}^m \Lambda s_i \geq \sum_{i=1}^m L_i^\star(\sigma) \cdot s_i = \sum_{j'=1}^n p_{j'} > \sum_{j'=1}^{j-1} p_{j'}$. This means that there must be a machine with load less than $\Lambda$ at the start of step $j$, since otherwise $\sum_{j'=1}^{j-1} p_{j'} = \sum_{i=1}^m L_i(\sigma_{\leq j-1}) \cdot s_i \geq \sum_{i=1}^m \Lambda s_i$. We may thus define $f = \max\{i \in \{1, \ldots, m\} \colon L_i(\sigma_{\leq j-1}) \leq \Lambda\}$ to be the fastest machine that is not overloaded (see Figure 13).

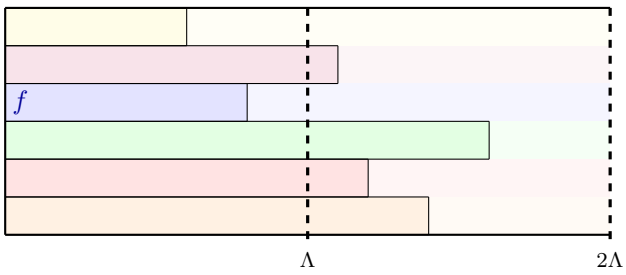


Figure 13: Sketch of the proof of Lemma 8.6. The machine $f$ is the fastest machine with load at most $\Lambda$. Since $\textsc{Opt}(\sigma) \leq \Lambda$, SlowFit is processing at least one job on machines $\{f+1, \ldots, m\}$ that Opt is not. But this job would fit on machine $f$, which is a contradiction with the fact that SlowFit puts each job on the slowest machine where it fits.

If $f = m$, then

$$L_m(\sigma_{\leq j-1}) + p_j/s_m \leq \Lambda + \textsc{Opt}(\sigma) \leq 2\Lambda,$$

which contradicts the fact that the subroutine of SlowFit doesn't find a machine for job $j$. Hence, we can assume $f < m$.

Since all machines $i > f$ have load $L_i(\sigma_{\leq j-1}) > \Lambda$, we get

$$\begin{aligned}\sum_{i=f+1}^{m} L_i(\sigma_{\leq j-1}) \cdot s_i &> \sum_{i=f+1}^{m} \Lambda s_i \\ &\geq \sum_{i=f+1}^{m} \textsc{Opt}(\sigma) \cdot s_i \\ &\geq \sum_{i=f+1}^{m} L_i^\star(\sigma) \cdot s_i.\end{aligned}$$

This means that there must be a job $j' < j$ that SlowFit assigns to a machine $i_{j'} \in \{f+1, \ldots, m\}$ and that Opt assigns to a slower machine $i^\star \in \{1, \ldots, f\}$. Then, $p_{j'}/s_{i^\star} \leq \textsc{Opt}(\sigma) \leq \Lambda$. Consider the moment when SlowFit assigned job $j'$ to machine $i_{j'}$. At this point, we had

$$L_f(\sigma_{\leq j'-1}) + \frac{p_{j'}}{s_f} \leq L_f(\sigma_{\leq j-1}) + \frac{p_{j'}}{s_{i^\star}} \leq 2\Lambda.$$

But then, SlowFit could have assigned job $j'$ to machine $f$, which contradicts the fact that $j'$ was assigned to a machine $i_{j'} > f$. □

**Corollary 8.7** (Aspnes et al. [1997])**.** SlowFit *is strictly* 8*-competitive for the online load balancing problem with related machines.*

*Proof.* The claim follows from Lemmas 8.5 and 8.6 if we initially set $\Lambda_0 = p_1/s_m \leq \textsc{Opt}(\sigma)$. □

*Remark* 8.8. Berman et al. [2000] presented the currently best known deterministic algorithm for online load balancing on related machines which is 5.828-competitive as well as the best known lower bound of 2.438.

## 8.3 Restricted assignment

So far, we assumed that the difference in processing times on different machines is consistent between jobs, i.e., that we can order machines from slow to fast. We now want to investigate how much harder online load balancing becomes without this property. To do this, we consider load balancing with *restricted assignment*, where every job has a fixed processing time but cannot be processed on all machines, i.e., $p_{ij} \in \{p_j, \infty\}$ for all machines $i$ and jobs $j$. This setting turns out to be substantially harder.

**Theorem 8.9** (Azar et al. [1995])**.** *Every deterministic online algorithm for load balancing with restricted assignment has competitive ratio at least* $\log m$*.*

*Proof.* Let Alg be any fixed online algorithm. We take $k \in \mathbb{N}$ and construct an adversarial request sequence $\sigma = (1, 2, \ldots, m-1)$ for $m = 2^k$ machines and with $p_{ij} \in \{1, \infty\}$ for all jobs $j \in \{1, \ldots, m-1\}$ (see Figure 14). If we let $M_j = \{i \in \{1, \ldots, m\} : p_{ij} = 1\}$ denote the machines that can run job $j$, we need to specify $M_j$ for all $j \in \{1, \ldots, m-1\}$ to fully describe the instance. For $j \in \{1, \ldots, m/2\}$ we set $M_j = \{j, j + m/2\}$. By symmetry, we can assume that Alg schedules each job $j \in \{1, \ldots, m/2\}$ on machine $j$, otherwise we rename machines to ensure this is the case. Similarly, we let $M_{j'+m/2} = \{j', j' + m/4\}$ for $j' \in \{1, \ldots, m/4\}$ and assume that Alg schedules each job $j \in \{m/2 + 1, \ldots, m/2 + m/4\}$ on machine $j - m/2$, and so on. Eventually, for all $x \in \{0, \ldots, k\}$ we have $M_{j'+X_x} = \{j', j' + m/2^x\}$ for $j' \in \{1, \ldots, m/2^x\}$ and $X_x := \sum_{l=1}^{x-1} \frac{m}{2^l}$, and job $j' + X_x$ is scheduled on machine $j'$. The number of jobs scheduled on machine 1 is $k = \log m$, hence $\textsc{Alg}(\sigma) = \log m$.

On the other hand, we can put a single job on every machine by assigning job $j' + X_x$ to machine $j' + m/2^x$ for every $x \in \{0, \ldots, k\}$ and every $j' \in \{1, \ldots, m/2^x\}$. Hence $\textsc{Opt}(\sigma) = 1$ and $\textsc{Alg}(\sigma) \geq \log m \cdot \textsc{Opt}(\sigma)$, thus the strict competitive ratio of Alg is at least $\log m$.

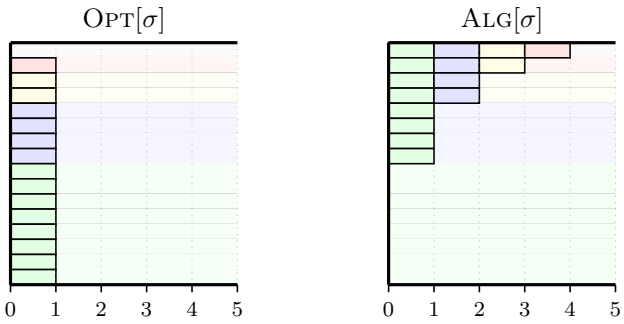


Figure 14: Illustration of the proof of Theorem 8.9. Jobs arrive in the order green, blue, yellow, red. Green jobs can be processed by all machines, blue jobs only by the first half of the machines, yellow jobs by the first quarter of the machines, etc.

By increasing the number of machines, we can repeat this construction an arbitrary number of times on different machines. We can then apply Proposition 1.12 to obtain the claimed bound. □

We claim that the simple greedy algorithm (Greedy) that puts every job $j$ on the least loaded machine $i$ with $p_{ij} \neq \infty$ is almost optimal in terms of competitive ratio.

**Theorem 8.10** (Azar et al. [1995])**.** Greedy *is strictly* $(\lceil \log m \rceil + 1)$*-competitive for online load balancing with restricted assignment.*

*Proof.* We consider an input sequence $\sigma$ and let $R_{\lambda,i} := \max\{0, L_i(\sigma) - \lambda \textsc{Opt}(\sigma)\}$ denote the load on machine $i$ after time $\lambda \textsc{Opt}(\sigma)$ for $\lambda \in \mathbb{N}$, and $R_\lambda := \sum_{i=1}^m R_{\lambda,i}$. Note that, by definition, $R_0 = \sum_{j=1}^n p_j$. **We claim: $R_{\lambda+1} \leq R_\lambda/2$ for all $\lambda \in \mathbb{N}$.** With this claim, for all $\lambda \in \mathbb{N}$, we have $R_\lambda \leq R_0/2^\lambda$ and thus

$$\textsc{Opt}(\sigma) \geq \frac{1}{m} \sum_{j=1}^n p_j = \frac{1}{m} R_0 \geq \frac{2^\lambda}{m} R_\lambda,$$

and, in particular,

$$\textsc{Opt}(\sigma) \geq \frac{2^{\lceil \log m \rceil}}{m} R_{\lceil \log m \rceil} \geq R_{\lceil \log m \rceil}.$$

But this means that even if the total load $R_{\lceil \log m \rceil}$ after time $\lceil \log m \rceil \cdot \textsc{Opt}(\sigma)$ goes on a single machine, Greedy must finish before time $(\lceil \log m \rceil + 1) \cdot \textsc{Opt}(\sigma)$, which proves the theorem.

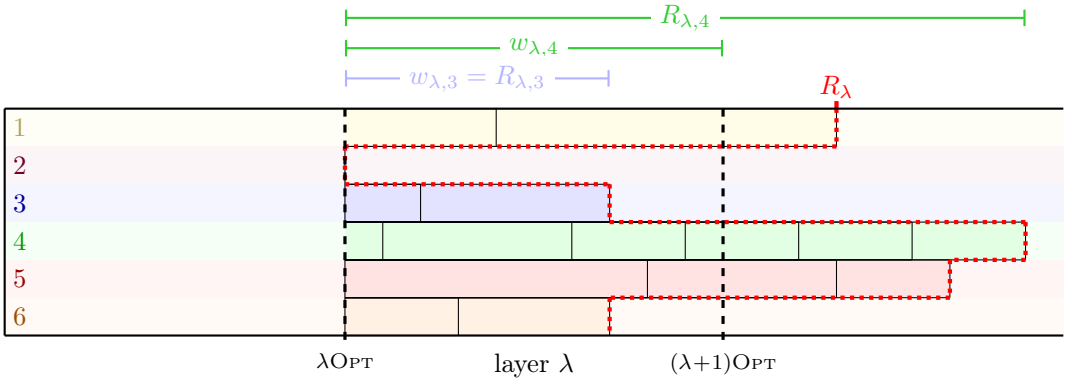


Figure 15: Illustration of the definitions in the proof of Theorem 8.10.

It remains to establish our claim that $R_{\lambda+1} \leq R_\lambda/2$ for all $\lambda \in \mathbb{N}$. Consult Figure 15 with the following. We show $w_\lambda := R_\lambda - R_{\lambda+1} \geq R_{\lambda+1}$, which immediately implies $R_\lambda = w_\lambda + R_{\lambda+1} \geq 2R_{\lambda+1}$, as claimed. We define *layer* $\lambda$ to be the time period $[\lambda \textsc{Opt}(\sigma), (\lambda+1)\textsc{Opt}(\sigma))$ and let $w_{\lambda,i} := R_{\lambda,i} - R_{\lambda+1,i} \leq \textsc{Opt}(\sigma)$ be the load on machine $i$ in layer $\lambda$. Let $r_{\lambda,j} := \max\{0, \min\{p_j, C_j - \lambda\textsc{Opt}(\sigma)\}\}$ denote the remaining processing time of job $j$ at the start of layer $\lambda$. We further let $J_i^\star$ be the set of jobs scheduled on machine $i$ by Opt and $R^\star_{\lambda,i} := \sum_{j \in J_i^\star} r_{\lambda,j} \leq \textsc{Opt}(\sigma)$. It is sufficient to show $w_{\lambda,i} \geq R^\star_{\lambda+1,i}$ for all machines $i$, because then $w_\lambda = \sum_{i=1}^m w_{\lambda,i} \geq \sum_{i=1}^m R^\star_{\lambda+1,i} = R_{\lambda+1}$. This is obviously true if $w_{\lambda,i} = \textsc{Opt}(\sigma)$, because $R^\star_{\lambda+1,i} \leq \textsc{Opt}(\sigma)$, or if $R^\star_{\lambda+1,i} = 0$.

Now take a machine $i$ with $w_{\lambda,i} < \textsc{Opt}(\sigma)$ and $R^\star_{\lambda+1,i} > 0$. The latter allows us to take any job $j^\star \in J_i^\star$ that is still unfinished after layer $\lambda$, i.e., with $r_{\lambda+1,j^\star} > 0$. This job is not scheduled on machine $i$, because $w_{\lambda,i} < \textsc{Opt}(\sigma)$ and thus $w_{\lambda',i} = 0$ for all $\lambda' > \lambda$. By definition of Greedy, the starting time of $j^\star$ is $C_{j^\star} - p_{j^\star} \leq \lambda\textsc{Opt}(\sigma) + w_{\lambda,i}$, because otherwise Greedy would have put $j^\star$ on machine $i$. It follows that

$$\begin{aligned} R^\star_{\lambda+1,i} &= r_{\lambda+1,j^\star} + \sum_{j \in J_i^\star \setminus \{j^\star\}} r_{\lambda+1,j} \\ &\leq r_{\lambda+1,j^\star} + \textsc{Opt}(\sigma) - p_{j^\star} \\ &\leq C_{j^\star} - (\lambda+1)\textsc{Opt}(\sigma) + \textsc{Opt}(\sigma) - p_{j^\star} \\ &\leq w_{\lambda,i}. \end{aligned}$$

□

## 8.4 Unrelated machines

Finally, we turn to a setting where each job can have arbitrary processing times on the different machines. This is often referred to as the *unrelated machines* setting. We present an elegant algorithm Pdlb for this setting that relies on the doubling technique (Lemma 8.5) together with a primal-dual approach for the necessary subroutine.

Recall that Lemma 8.5 requires a subroutine that has an input parameter $\Lambda$ and must always compute a solution of bounded quality when provided with a valid bound $\Lambda \geq \textsc{Opt}(\sigma)$. The subroutine of Pdlb is based on an LP-relaxation (PLB) for the problem of distributing the maximum number of jobs on the machines while not exceeding load $\Lambda$ (see below). In the underlying integer program, the variable $x_{ij} \in \{0,1\}$ indicates whether job $j$ is scheduled on machine $i$. Since, for $\Lambda \geq \textsc{Opt}(\sigma)$, we know that $\textsc{Opt}[\sigma]$ schedules each job $j$ on a machine $i$ with $p_{ij} \leq \Lambda$, we can restrict our variables $x_{ij}$ accordingly. To that end, we define $S := \{(i,j) \in \{1,\dots,m\} \times \{1,\dots,n\} : p_{ij} \leq \Lambda\}$.

Observe that the integer program has a solution of value $n$ (i.e., all jobs can be scheduled) if and only if $\Lambda \geq \textsc{Opt}(\sigma)$. Consequently, if $\Lambda \geq \textsc{Opt}(\sigma)$, the optimum solution of the LP-relaxation (PLB) must have value $n$ (the converse may not be true). By weak duality (Theorem 7.1), the dual (DLB) can therefore not have a solution of value below $n$ if $\Lambda \geq \textsc{Opt}(\sigma)$.

(PLB)

| | | |
|---|---|---|
| max | $\sum_{(i,j)\in S} x_{ij}$ | |
| s.t. | $\sum_{j:(i,j)\in S} p_{ij}x_{ij} \leq \Lambda$, | $\forall i \in \{1,\dots,m\}$ |
| | $\sum_{i:(i,j)\in S} x_{ij} \leq 1$, | $\forall j \in \{1,\dots,n\}$ |
| | $x_{ij} \geq 0$ | $\forall (i,j) \in S$ |

(DLB)

| | | |
|---|---|---|
| min | $\Lambda \sum_{i=1}^m y_i + \sum_{j=1}^n z_j$ | |
| s.t. | $z_j + p_{ij}y_i \geq 1$, | $\forall (i,j) \in S$ |
| | $y_i \geq 0$ | $\forall i \in \{1,\dots,m\}$ |
| | $z_j \geq 0$ | $\forall j \in \{1,\dots,n\}$ |

The idea behind the subroutine of Pdlb is a bit different from the primal-dual technique in Chapter 7. We again maintain a feasible dual solution, but we do not use it to bound the quality of our primal solution. In fact, the subroutine always schedules all jobs and may not maintain a feasible primal solution at all. Instead, we use the dual solution as a means to detect when to abort: If the dual objective value ever falls below $n$, we may conclude that $\Lambda < \textsc{Opt}(\sigma)$ and abort. The subroutine is specified formally below.

**Algorithm 1:** PDLB subroutine

**for** $i \in \{1, \dots, m\}$ **do**
  $y_i \leftarrow 1/2m$
**foreach** *job* $j$ *arriving online* **do**
  **if** $\min_{i \in \{1,\dots,m\}} p_{ij} > \Lambda$ **or** $\max_{i \in \{1,\dots,m\}} y_i > 1$ **then**
    **abort**
  $i \leftarrow \arg\min_{i':(i',j) \in S} p_{i'j} y_{i'}$
  $x_{ij} \leftarrow 1$
  $z_j \leftarrow 1 - p_{ij} y_i$
  $y_i \leftarrow y_i(1 + p_{ij}/2\Lambda)$

We now show that the subroutine behaves as desired.

**Lemma 8.11.** *Given* $\Lambda \geq \text{OPT}(\sigma)$*, the subroutine of* PDLB *computes a solution of load at most* $\Lambda \cdot \mathcal{O}(\log m)$*.*

*Proof.* We first show that, if the subroutine does not abort, it computes a feasible solution where each machine has load at most $\Lambda \cdot \mathcal{O}(\log(m))$. Note that we do not claim that the subroutine computes a feasible primal solution, but allow it to violate $\sum_{j=1}^{n} p_{ij} x_{ij} \leq \Lambda$.

In case the subroutine does not abort, we argue that the produced dual solution fulfills $y_i \leq 3/2$ for all $i \in \{1, \dots, m\}$, by induction over the number of jobs assigned to machine $i$. This is clear if no jobs get assigned, since the initial value $y_{i,0}$ of $y_i$ is $1/2m < 3/2$. Now assume that at least one job is assigned to machine $i$ and consider the step in which the last job gets assigned. Since the algorithm does not abort, we have $y_i \leq 1$ at the beginning of this step, and by choice of $i$ with $(i, j) \in S$ we have $p_{ij} \leq \Lambda$ (note that such an $i$ must exists, since the algorithm did not abort). This means that $y_i$ is updated to $y_i(1 + p_{ij}/2\Lambda) \leq 3/2$ and the claim is true.

Now, recall that $J_i(\sigma) \subseteq \{1, \dots, n\}$ denotes the set of jobs assigned to machine $i$. Since $y_{i,0} = 1/2m$ and we update $y_i$ each time a job gets assigned to machine $i$, we have

$$y_i = \frac{1}{2m} \prod_{j \in J_i(\sigma)} (1 + p_{ij}/2\Lambda) \overset{p_{ij} \leq \Lambda}{\geq} \frac{1}{2m} \prod_{j \in J_i(\sigma)} \left(\frac{3}{2}\right)^{p_{ij}/\Lambda},$$

where we used $1 + x/2 \geq (3/2)^x$ for $x \in [0, 1]$.

Combining this with $y_i \leq 3/2$ and taking logarithms, we get

$$\log \frac{3}{2} \geq \log \frac{1}{2m} + \sum_{j \in J_i(\sigma)} \frac{p_{ij}}{\Lambda} \log \frac{3}{2},$$

and thus

$$L_i(\sigma) = \sum_{j \in J_i(\sigma)} p_{ij} \leq \Lambda \left( \frac{\log(3/2) + \log(2m)}{\log(3/2)} \right) = \Lambda \cdot \frac{\log(3m)}{\log(3/2)} = \Lambda \cdot \mathcal{O}(\log(m)),$$

as claimed.

It remains to show that if the subroutine of PDLB aborts, then our guess of $\text{OPT}(\sigma)$ must have been too small, i.e., $\Lambda < \text{OPT}(\sigma)$. First observe that if the algorithm aborts because $\min_{i \in \{1,\dots,m\}} p_{ij} > \Lambda$, we clearly have $\text{OPT}(\sigma) > \Lambda$, since scheduling job $j$ alone already produces a load of more than $\Lambda$. Secondly, observe that if $\Lambda \geq \text{OPT}(\sigma)$, the primal LP has a solution of value $n$, since all jobs can be scheduled.

We now assume that the algorithm aborts because $\max_{i \in \{1,\dots,m\}} y_i > 1$ in some iteration $j$. It suffices to show that in this case the dual optimum has value less than $n$. By weak duality, it then

follows that the primal optimum must also have value less than $n$, which implies that $\Lambda < \textsc{Opt}$, since not all jobs can be scheduled.

Do this end, observe that the algorithm maintains a dually feasible solution: Since $z_j$ is set to $1 - p_{ij}y_i$ with $i$ being the machine that minimizes $p_{ij}y_i$, the constraints involving $z_j$ are satisfied in step $j$, and, since $y_i$ is non-decreasing, they remain satisfied later on.

To bound the value of the dual optimum, consider a step $j$ where the algorithm assigns job $j$ to some machine $i$. This increases the value of $y_i$ by $\Delta_{ij} := y_i - y_i' = p_{ij}y_i'/2\Lambda$, where $y_i'$ is the value of $y_i$ before the update. Overall, the dual objective function increases by $(1-p_{ij}y_i')+p_{ij}y_i'/2 = 1-\Lambda\cdot\Delta_{ij}$ due to the increase of $z_j$ and $y_i$. This means that the dual objective function of the computed solution can be written as

$$\begin{aligned}
\Lambda\sum_{i=1}^{m} y_i + \sum_{j=1}^{n} z_j &= \Lambda\sum_{i=1}^{m} y_{i,0} + \sum_{i=1}^{m}\sum_{j\in J_i(\sigma)} (1-\Lambda\cdot\Delta_{ij}) \\
&= \Lambda\sum_{i=1}^{m} y_{i,0} + n - \Lambda\cdot\sum_{i=1}^{m}(y_i - y_{i,0}) \\
&= 2\Lambda\sum_{i=1}^{m}\frac{1}{2m} + n - \Lambda\sum_{i=1}^{m} y_i \\
&= \Lambda + n - \Lambda\sum_{i=1}^{m} y_i.
\end{aligned}$$

By assumption, the algorithm aborts because there exists a machine $i$ with $y_i > 1$. But, with $y_i \geq 0$, this implies that the objective function value of the current, feasible dual solution $y, z$ is less than $n$, and therefore the same is true for the optimum primal solution. We therefore can conclude that $\Lambda < \textsc{Opt}(\sigma)$. □

Using Lemma 8.11, we can now apply the doubling technique (with $\Lambda_0 = \min_i p_{i1}$) and obtain the following result.

**Theorem 8.12** (Buchbinder and Naor [2011])**.** Pdlb *is $\mathcal{O}(\log m)$-competitive for load balancing.*

*Remark* 8.13. The bound in Theorem 8.12 was first proven by Aspnes et al. [1997] using a different approach. The algorithm Pdlb is a primal-dual interpretation of one of their algorithms.

# Bibliography


S. Albers. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27(3):682–693, 1998.

S. Albers, B. von Stengel, and R. Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. volume 56, pages 135–139, 1995.

J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *Journal of the ACM*, 44(3):486–504, 1997.

Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. *Journal of Algorithms*, 18(2):221–237, 1995.

L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404–411, 1985.

P. Berman, M. Charikar, and M. Karpinski. On-line load balancing for related machines. *Journal of Algorithms*, 35(1):108–121, 2000.

A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. *Journal of the ACM*, 39:745–763, 1992.

N. Buchbinder and J. Naor. Fair online load balancing. *Journal of Scheduling*, 16(1):117–127, 2011.

Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9:91–103, 1988.

M. Chrobak and L. L. Larmore. On fast algorithms for two servers. *Journal of Algorithms*, 12(4): 607–614, 1991a.

M. Chrobak and L. L. Larmore. An optimal on-line algorithm for k servers on trees. *SIAM Journal on Computing*, 20(1):144–148, 1991b.

M. Chrobak, H. Karloff, T. Payne, and S. Vishwnathan. New ressults on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.

A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

R. Fleischer and M. Wahl. Online scheduling revisited. *Journal of Scheduling*, 3:343–353, 2000.

R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

S. Irani. Two results on the list update problem. *Information Processing Letters*, 38(6):301–306, 1991.

S. Irani and R. Rubinfeld. A competitive 2-server algorithm. *Information Processing Letters*, 39(2): 85–91, 1991.

J. R. Isbell. Finitary games. *Contributions to the Theory of Games*, III:79–96, 1957.

A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, 1988.

R. Karp and P. Raghavan, 1990. Personal communication reported by Irani [1991].

R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd annual ACM symposium on Theory of computing (STOC)*, pages 352–358, 1990.

J. M. Kleinberg. A lower bound for two-server balancing algorithms. *Information Processing Letters*, 52(1):39–43, 1994.

E. Koutsoupias and C. H. Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42(5): 971–983, 1995.

M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1-6):816–825, 1991.

N. Reingold, J. Westbrook, and D. D. Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994.

J. F. Rudin III and R. Chandrasekaran. Improved bounds for the online scheduling problem. *SIAM Journal on Computing*, 32:717–735, 2003.

D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

B. Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47 (1):5–9, 1993.

E. Torng. A unified analysis of paging and caching. *Algorithmica*, 20(2):175–200, 1998.

A. C.-C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.