Report

Online algorithms with advice

Author(s):
Böckenhauer, Hans-Joachim; Komm, Dennis; Královič, Rastislav; Královič, Richard; Mömke, Tobias

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-006733662 →

Online Algorithms with Advice*

Hans-Joachim Böckenhauer[1], Dennis Komm[1], Rastislav Královič[2], Richard Královič[1], and Tobias Mömke[1]

[1] Department of Computer Science, ETH Zurich, Switzerland
{hjb, dennis.komm, richard.kralovic, tobias.moemke}@inf.ethz.ch
[2] Department of Computer Science, Comenius University, Bratislava, Slovakia
kralovic@dcs.fmph.uniba.sk

Abstract. In online problems, the input forms a finite sequence of *requests*. Each request must be *processed*, i. e., a partial output has to be computed only depending on the requests having arrived so far, and it is not allowed to change this partial output subsequently. The aim of an *online algorithm* is to produce a sequence of partial outputs that optimizes some global measure. The most frequently used tool for analyzing the quality of online algorithms is the *competitive analysis* which compares the solution quality of an online algorithm to the optimal solution for the whole input sequence, and in fact measures the degradation in the solution quality caused by the lack of any information about the input.
In this paper, we investigate to what extent the solution quality can be improved by allowing the algorithm to extract a given amount of information about the input. We consider the recently introduced notion of *advice complexity* where the algorithm, in addition to being fed the requests one by one, has access to a tape of advice bits that were computed by some *oracle* function from the complete input. The advice complexity is the number of advice bits read.
We introduce an improved model of advice complexity and investigate the connections of advice complexity to the competitive ratio of both deterministic and randomized online algorithms using the paging problem, job shop scheduling, and the routing problem on a line as sample problems.
Our results for all of these problems show that very small advice (only three bits in the case of paging) already suffices to significantly improve over the best deterministic algorithm. Moreover, to achieve the same competitive ratio as any randomized online algorithm, a logarithmic number of advice bits is sufficient. On the other hand, to obtain optimality, much larger advice is necessary.

1 Introduction

Many problems such as routing, scheduling, or the paging problem work in so-called online environments and their algorithmic formulation and analysis demand a model in which an algorithm that deals with such a problem knows only a part of its input at any specific point during runtime. These problems are called *online problems* and the respective algorithms are called *online algorithms*. In such an online setting, an online algorithm $\mathtt{A}$ has a huge disadvantage compared to offline algorithms (i. e., algorithms knowing the whole input already at the beginning of their computation) since $\mathtt{A}$ has to make decisions at any time step i without knowing what the next chunk of input at time step $i+1$ will be. As $\mathtt{A}$ has to produce a part of the final output in every step, it cannot revoke decisions it has already made. These decisions can only be made by merely taking input chunks from time steps 1 to i into account and maybe applying some randomization.

The output quality of such an online algorithm is often analyzed by the well-established *competitive ratio* introduced by Sleator and Tarjan in [8]. Informally speaking, the output

* This work was partially supported by ETH grant TH 18 07-3, and APVV grant 0433-06.

quality of an online algorithm A is measured by comparing it to an optimal offline algorithm. For an online problem P, let $\textsc{Opt}(I)$ denote an optimal offline solution for a certain input I of P. Then, for some $c \geq 0$, A is called *c-competitive* if there exists some constant $\alpha \geq 0$ such that, for any such input sequence I, $cost(\mathtt{A}(I)) \leq c \cdot cost(\textsc{Opt}(I)) + \alpha$.

As a rather powerful tool, randomness is often employed in the design of online algorithms. The computations (sometimes also called *decisions*) of a randomized online algorithm R hereby heavily depend on a sequence of random bits often viewed as the content of a random tape accessible by R. For a fixed content of the random tape, R then behaves deterministically and achieves a certain competitive ratio. The expected value of the competitive ratio over all possible contents of the random tape is then used to measure the quality of a randomized online algorithm.

On the downside, it seems rather unfair to compare online and offline algorithms since there is simply no reasonable application of an offline algorithm in an online environment. Therefore, offline algorithms are in general more powerful than online algorithms.

Hence, we are interested not only in comparing the output quality of A to that of an optimal offline algorithm B, but we want to investigate what amount of information A really lacks. Surprisingly (and as already discussed in [4]), there are problems where only one straightforward piece of information (i. e., one bit) is needed for allowing A to be as good as B, e. g., the problem SkiRental, see [2, 4]. Clearly, this does not hold in general and thus we are interested in a formal framework allowing us to classify online problems according to how much information about the future input parts is needed for solving them optimally or with a specific competitive ratio. One way of measuring the amount of information needed for an online algorithm to be optimal is called *advice complexity* and was proposed in [4]. This model of advice complexity can informally be viewed as a cooperation of a deterministic online algorithm A and an *oracle* O, which may passively communicate with A. The oracle O, which has unlimited computational power, sees the whole input of A in advance and writes bitwise information needed by A onto an *advice tape* before A reads any input. Then, A can access the bits from the advice tape in a sequential manner, just as a randomized algorithm would use its random tape. The *advice complexity* of A on an input I is now defined as the number of advice bits A reads while processing this input. As usual, we consider the advice complexity as a function of the input size n by taking the maximum value over all inputs of length at most n.

Note that, by our definition, the oracle has no possibility to explicitly indicate the end of the advice. It must provide as much advice as asked by the algorithm. This eliminates the ability of the oracle to encode some information into the length of the advice string, as it was the case in [4]. As a result, our model is cleaner and more consistent with other complexity measures like, e. g., the number of random bits required for randomized algorithms.

Besides asking for the amount of advice that is necessary to compute an optimal solution, we also deal with the question whether some small advice might help to significantly reduce the competitive ratio. We analyze the advice complexity of three different online problems, namely the well-known paging problem, a job shop scheduling problem and the problem of routing communication requests in a line-topology network. Using these sample problems, we investigate the relation between advice complexity and the competitive ratio of deterministic as well as randomized algorithms.

We now give an overview of the results in this paper. We start with the formal definition of our model and some general observations in Section 2.

As a first example of a notoriously hard online problem, we consider the well-known *paging problem* in Section 3 and show that very little advice is needed to significantly improve over every deterministic online algorithm. Recall that, for the paging problem, we are given a fast memory, called *buffer* with a capacity of K data units, called *pages*, and a large, but slow secondary memory. An input now consists of a sequence of n page requests. Whenever a requested page is not in the buffer, it has to be fetched from the secondary memory. This situation is called a *page fault*, and the goal is to minimize the number of page faults. It is well-known that any deterministic online algorithm for the paging problem cannot achieve a better competitive ratio than K, and even randomized algorithms can at best achieve a ratio of H_K, where H_K is the K-th harmonic number. As for our advice complexity model, we show that one advice bit per request suffices to compute an optimal solution, and this bound is asymptotically tight for large K. Furthermore, we give upper and lower bounds on the number of required advice bits to achieve any given competitive ratio. In particular, these bounds imply the quite surprising result that a total of 3 advice bits suffices to improve the competitive ratio by a factor of 2 (for large enough K) in comparison with any deterministic algorithm. Furthermore, $\log K$ bits of advice are sufficient to achieve a competitive ratio asymptotically equal to the ratio of the best randomized algorithm.

Another example of a hard online problem, which we consider in Section 4, is the *disjoint path allocation problem*. For this problem, we have given a communication network with line topology and want to satisfy a maximum number of communication requests which arrive in an online fashion, under the constraint that the communication paths corresponding to the requests have to be edge-disjoint. It is known that, for n requests, no deterministic algorithm can achieve a better competitive ratio than $n/2$. We show that randomization also does not help too much for this problem, by proving a lower bound of $n/6 - \mathcal{O}(1)$ on the competitive ratio. Similarly as for the paging problem, one advice bit per request is sufficient to obtain an optimal algorithm. Furthermore, we present an algorithm with advice that achieves a competitive ratio of r using at most $(\frac{n}{r} + 3) \log n + \mathcal{O}(1)$ advice bits. On the other hand, in order to achieve a competitive ratio of r, at least $\frac{n+2}{2r} - 2$ advice bits are needed. Instead of using the number of requests, the complexity of this problem can also be measured with respect to the number of vertices of the underlying line network. Any algorithm for the problem needs many advice bits to compute an optimal solution also for this measure.

Section 5 is devoted to the *job shop scheduling* problem with two jobs and unit-length tasks. In this problem, we want to schedule two jobs, consisting of n unit-length tasks each, on n machines such that each job is processed exactly once by each machine in a prespecified order. If two jobs want to access the same machine within one time step, one of two jobs has to wait for one time step; the goal is to minimize the number of these delays. It is well-known that no deterministic online algorithm for this problem can achieve a competitive ratio better than $9/8 - \varepsilon$ (see [7]). We show that an online algorithm with advice can compute an optimal solution using $\lceil \sqrt{n} \rceil$ advice bits and prove an asymptotically matching lower bound on the number of advice bits. Furthermore, $\mathcal{O}(\log n)$ advice bits are sufficient to obtain a competitive ratio of $1 + \mathcal{O}(1/\sqrt{n})$, which is as good as the best known randomized algorithm for this job shop scheduling problem.

To sum up, the presented results for all three analyzed problems show some consistent features: Small advice (i. e., $\mathcal{O}(\log n)$ bits) is sufficient to significantly improve the competitive ratio over the best deterministic algorithm; our results imply that, for the paging problem, even constant advice is sufficient. Furthermore, small advice is sufficient to be on par with the best randomized algorithms; we have proven this for all three analyzed problems. On

the other hand, to obtain optimal algorithms, much larger advice ($\Omega(n)$ bits for paging and disjoint path allocation, $\Omega(\sqrt{n})$ bits for job shop scheduling) is necessary.

2 Preliminaries

Before we present our results, we formally define the terms of online algorithms and competitive analysis as introduced in [8].

Definition 1. Consider an input sequence $I = (x_1, \ldots, x_n)$. An *online algorithm* $\mathtt{A}$ computes the output sequence $\mathtt{A}(I) = (y_1, \ldots, y_n)$, where $y_i = f(x_1, \ldots, x_i)$ for some function f. The *cost* of the solution is given by a function $C(\mathtt{A}(I))$. By $\mathcal{A} = \mathtt{A}(I)$ we denote a solution computed by $\mathtt{A}$ on I.

An online algorithm is *c-competitive*, for some $c \geq 1$, if there exists a constant α such that, for each input sequence I, $C(\mathtt{A}(I)) \leq c \cdot C(\mathtt{OPT}(I)) + \alpha$, where $\mathtt{OPT}$ is an optimal offline algorithm for the problem. An online algorithm is *optimal* if it is 1-competitive with $\alpha = 0$.

Often, randomization is used in the design of online algorithms. Formally, randomized online algorithms can be defined as follows.

Definition 2. A randomized online algorithm $\mathtt{R}$ computes the output sequence $\mathcal{R}^\phi = \mathtt{R}^\phi(I) = (y_1, \ldots, y_n)$ such that y_i is computed from $\phi, x_1, \ldots, x_i$, where ϕ is the content of the random tape, i. e., an infinite binary sequence where every bit is chosen uniformly at random and independently of others. By $C(\mathtt{R}(I))$ we denote the random variable expressing the cost of the solution computed by $\mathtt{R}$ on I.

$\mathtt{R}$ is c-competitive[3] if there exists a constant α such that for each input sequence I, $E[C(\mathtt{R}(I))] \leq c \cdot C(\mathtt{OPT}(I)) + \alpha$.

Our work focusses on the model where the algorithm can use some information, called also *advice*, about the future input. We introduce the computation model in an analogous way to the model of randomized online algorithms.

Definition 3. An *online algorithm* $\mathtt{A}$ *with advice* computes the output sequence $\mathcal{A}^\phi = \mathtt{A}^\phi(I) = (y_1, \ldots, y_n)$ such that y_i is computed from $\phi, x_1, \ldots, x_i$, where ϕ is the content of the advice tape, i. e., an infinite binary sequence.

Algorithm $\mathtt{A}$ is *c-competitive with advice complexity* $s(n)$ if there exists a constant α such that, for every n and for each input sequence I of length at most n, there exists some ϕ such that $C(\mathtt{A}^\phi(I)) \leq c \cdot C(\mathtt{OPT}(I)) + \alpha$ and at most $s(n)$ bits of ϕ have been accessed during the computation of $\mathtt{A}^\phi(I)$.

For the ease of notation, for both randomized online algorithms and online algorithms with advice, we write $\mathtt{A}(I)$ instead of $\mathtt{A}^\phi(I)$ if ϕ is clear from the context.

If $\mathtt{A}$ accesses b bits of the advice tape during some computation, we say that b *advice bits are communicated* to $\mathtt{A}$, or that $\mathtt{A}$ *uses* b *bits of advice*. The advice complexity of $\mathtt{A}$ gives an upper bound on the number of communicated advice bits, depending on the size n of the input.

[3] The notion of competitiveness for randomized online algorithms as used in this paper is called competitiveness against an *oblivious adversary* in the literature. For an overview of the different adversary models, see, e. g., [2].

Note that, if there exists a randomized online algorithm for some online problem achieving a competitive ratio r and using b random bits, the same competitive ratio r can be achieved by an online algorithm with advice using b advice bits.

We use $\log(x)$ to denote the logarithm of x with base 2. When proving upper bounds on advice complexity, we sometimes use the following idea. The oracle needs to communicate some n-bit string to the algorithm, but this string always contains only few ones or only few zeros. The following lemma describes an efficient encoding of such strings.

Lemma 1 *Consider an online algorithm* A *with advice complexity* $s_{\mathtt{A}}(n) = n$ *and competitive ratio* r*. Furthermore, assume that, for every input instance,* A *achieves competitive ratio* r *while using some* n*-bit advice string that contains at most* n/t *zeros or at least* $n - n/t$ *zeros, where* $t \geq 2$ *is a fixed constant. Then, it is possible to design an improved online algorithm* B *that knows the parameter* t *and achieves an advice complexity of*

$$s(n) = n \log \left(\frac{t}{(t-1)^{\frac{t-1}{t}}} \right) + 3 \log n + \mathcal{O}(1)$$

or

$$s(n) = \left(\frac{n}{t} + 3 \right) \log n + \mathcal{O}(1).$$

Proof. Let S be the number of all possible n-bit strings with at most n/t or at least $n - n/t$ zeros. To communicate any such string ϕ to the algorithm B, the oracle writes a number from the range $0 \ldots S-1$ indicating the position of ϕ in lexicographical ordering of all possible strings. To do so, $\lceil \log S \rceil$ bits are sufficient. The algorithm, however, needs to know n and t to be able to decode the advice. Since t is always known by the algorithm, it is sufficient to encode the value of n on the advice tape. This can be done by encoding the value $\lceil \log n \rceil$ in unary encoding, i. e., as $0^{\lceil \log n \rceil - 1}1$, at the beginning of the tape and afterwards encoding the value of n using $\lceil \log n \rceil$ bits. Hence, $\lceil \log S \rceil + 2\lceil \log n \rceil = \log S + 2 \log n + \mathcal{O}(1)$ bits of advice are sufficient to achieve r-competitiveness.

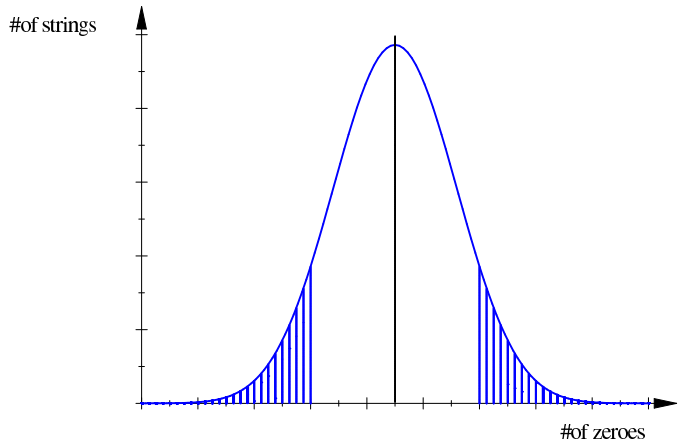


Fig. 1. Encoding strings with at most $n/(r+1)$ or at least $nr/(r+1)$ zeros

Now let us estimate the number S of possible strings (see Fig. 1). Since $\frac{(t-1)n}{t} = n - \frac{n}{t}$, we get that

$$S \leq 2 \sum_{i=0}^{\lfloor \frac{n}{t} \rfloor} \binom{n}{i} \leq 2 + 2\frac{n}{t} \binom{n}{\lfloor \frac{n}{t} \rfloor}$$

and subsequently

$$\log S \leq \mathcal{O}(1) + \log n + \log \binom{n}{\lfloor \frac{n}{t} \rfloor}.$$

Substituting $q := \lfloor \frac{n}{t} \rfloor$ and using Stirling's formula (we assume $q \geq 1$, otherwise the claim is trivial), we get that

$$\begin{aligned} \log \binom{n}{\lfloor \frac{n}{t} \rfloor} = \log \binom{n}{q} = \log \frac{n!}{q!(n-q)!} &= \log \left[\frac{\sqrt{n}}{\sqrt{2\pi q(n-q)}} \cdot \frac{n^n}{q^q (n-q)^{n-q}} \right] + \mathcal{O}(1) \\ &\leq \log \frac{n^n}{q^q (n-q)^{n-q}} + \mathcal{O}(1). \end{aligned}$$

It is straightforward to verify that $q^q(n-q)^{n-q}$ is decreasing in q for $q \in (0, n/2)$, so we have

$$\log \binom{n}{\lfloor \frac{n}{t} \rfloor} \leq \log \frac{n^n}{\frac{n}{t}^{\frac{n}{t}} (n - \frac{n}{t})^{n - \frac{n}{t}}} + \mathcal{O}(1) = n \log \frac{t}{(t-1)^{\frac{t-1}{t}}},$$

which concludes the proof of the first claim. To prove the second claim, notice that $\binom{n}{q} \leq n^q$, hence

$$\log S \leq \mathcal{O}(1) + \log n + \log(n^q) \leq \frac{n}{t} \log n + \log n + \mathcal{O}(1).$$

□

3 Paging

The memory of a computer can be arranged in a hierarchy that starts at the top with very fast but expensive kinds like the CPU's registers and its caches, has slow but cheap types of memory, such as the hard disc, at its bottom, and the *random access memory* (ram, often plainly called the *physical* or *main* memory) in between.

In the following, we focus on a type of memory hierarchy consisting only of two types of memory: the fast *physical memory* and the slow hard disc drive. To use memory as efficiently as possible, the technique of *paging* is widely used. The *physical memory* and the hard disc drive are subdivided into page frames of a fixed size s. Any program, on the other hand, only works on a virtual memory which consists of *logical pages* also of size s. The operating system then maps the logical pages to the page frames. If a program now wants to access a page that is currently not in the physical memory, but on the hard disc drive only, we call this a *page fault*. In this case, the operating system has to select some *victim* page frame in the physical memory, replace its content by the content of this page, and update the mapping of the virtual pages to the page frames. Obviously, the logical page contained by the victim frame before the page fault is no longer accessible in the physical memory. Selecting the victim frame is the main task of a paging algorithm. In what follows, we give a formal description.

Definition 4 (Paging Problem). The input is a sequence of integers representing requests to logical pages $I = (x_1, \ldots, x_n)$, $x_i > 0$. An online algorithm A maintains a buffer (content of the physical memory) $B = \{b_1, \ldots, b_K\}$ of K integers, where K is a fixed constant known to A. Before processing the first request, the buffer gets initialized as $B = \{1, \ldots, K\}$. Upon receiving a request x_i, if $x_i \in B$, then $y_i = 0$. If $x_i \notin B$, then a *page fault* occurs, and the algorithm has to find some *victim* b_j, i. e., $B := B \setminus \{b_j\} \cup \{x_i\}$, and $y_i = b_j$. The cost of the solution $\mathcal{A} = \mathtt{A}(I)$ is the number of page faults, i. e., $C(\mathcal{A}) = |\{y_i : y_i > 0\}|$.

Surely, here, we are not interested in consecutive appearances of the same page as part of the input and therefore, without loss of generality, we may assume $x_{i+1} \neq x_i$.

3.1 Known Facts about Paging

The paging problem (or PAGING for short) has been extensively studied, and many results are known for the deterministic settings. We recall the basic ones [2].

Fact 1 *Every deterministic algorithm is K-competitive or worse. There exists a K-competitive deterministic algorithm.*

Next, we provide a simple upper bound on the advice complexity of the optimal online algorithm with advice.

Fact 2 *There is an optimal online algorithm (i. e., a 1-competitive algorithm) solving* PAGING *with advice complexity n.*

Proof. Consider an input sequence I, and an optimal offline algorithm OPT processing it. In each step of OPT, call a page currently in the buffer *active* if it will be requested again before OPT replaces it by some other page. We design A such that in each step i, the set of OPT's active pages will be in B, and A will maintain with each page an *active* flag identifying this subset. If A gets an input x_i that causes a page fault, some *passive* (i. e., non-active) page is replaced by x_i. Moreover, A reads with each input also one bit from the advice tape telling whether x_i is active for OPT. Since the set of active pages is the same for OPT and A, it is immediate that A generates the same sequence of page faults. □

The following fact provides a lower bound for the competitive ratio of any randomized paging algorithm [2]:

Fact 3 *Every randomized algorithm is at least H_K-competitive, where H_K is the K-th harmonic number, i. e., $H_K = \sum_{i=1}^{K} \frac{1}{i}$.*

In our work, we sometimes use a class of paging algorithms, called *marking algorithms*, for proving upper bounds on algorithms with advice. This concept is also useful for providing upper bounds on the competitive ratio of randomized paging algorithms [2].

Definition 5. A *marking algorithm* works as follows:

1. Start with all pages unmarked.
2. Upon a hit, mark the page.
3. Upon fault, discard an unmarked page, and insert the new page as marked.
4. If there is no unmarked page in step 3, unmark all pages and repeat step 3.

The randomized marking algorithm selects the unmarked page in step 3 uniformly at random among all unmarked pages.

Fact 4 *The randomized marking algorithm is H_K-competitive, if the logical pages are chosen from a set of cardinality $K+1$, and $2H_K$-competitive in general.*

3.2 Constant Competitive Ratio

In Fact 2, we have shown that using one bit of advice per request is sufficient to obtain an optimal paging algorithm. In the next theorem, we show that it is possible to obtain paging algorithms with good competitive ratios using smaller advice. More precisely, we show an upper bound on the advice complexity depending on the competitive ratio, indicating that using an amortized constant amount of bits per request $s(n)/n < 1$ is enough to gain a constant competitive ratio.

Theorem 1 *For each constant $r \geq 1$, there exists an r-competitive algorithm for* Paging *with advice complexity*

$$s(n) = n \log \left(\frac{r+1}{r^{\frac{r}{r+1}}} \right) + 3 \log n + \mathcal{O}(1).$$

Proof. Consider an n-bit binary string ϕ defined as follows (comp. Fact 2): the i-th position is 0 if page x_i is removed from the buffer of the optimal offline algorithm after step i without requesting it in any following step, and 1 otherwise. Note that every zero in ϕ corresponds to a page fault in the optimal offline algorithm. Let ϕ contain k zeros. Now consider any string ϕ' obtained from ϕ by flipping at most $k(r-1)$ ones to zeros.

The algorithm `A` reads ϕ' from the advice tape and then works as follows: each request x_i (and also the corresponding page stored in memory after processing it) is labelled either *active*, if $\phi'[x_i] = 1$, and *passive* otherwise. Whenever a page fault occurs, the algorithm replaces a passive page.

First, we argue that the algorithm is well-defined, i. e., that upon each page fault there is at least one passive page in the memory. It is easy to see that if $\phi' = \phi$, the algorithm is valid, because the optimal algorithm must replace some page upon page fault. Moreover, since ϕ' was obtained from ϕ by flipping ones to zeros, it is easy to see that in each time step the set of active pages of the algorithm is a subset of active pages of the optimal algorithm. Hence, there is always some passive page.

Second, we argue that, for each place where ϕ and ϕ' differ, the algorithm makes only one additional fault. Consider an arbitrary page fault made by `A` that was not made by the optimal algorithm. Clearly, this page fault was caused by a page x_i that was in the memory of the optimal algorithm and not in the memory of `A`, meaning that the previous occurrence of x_i in the input had the original bit one flipped to zero in ϕ'. Hence, each additional page fault of the online algorithm can be charged to the flip of the most recent occurrence of the page.

Since ϕ contains k zeros, the optimal algorithm makes k faults. Hence, summarizing the previous arguments, one gets that in order to achieve a competitive ratio r, it is sufficient to supply the advice algorithm with any string ϕ' obtained from ϕ by flipping at most $k(r-1)$ ones to zeros. Indeed, in this case, the advice algorithm makes at most $k + k(r-1) = kr$ faults.

Consider an arbitrary string ϕ with k zeros. If $k \leq \frac{n}{r+1}$ or $k \geq \frac{rn}{r+1}$, let $\phi' = \phi$. Otherwise, flip arbitrary $k(r-1)$ ones, so ϕ' contains exactly $k + k(r-1) = rk \geq \frac{rn}{r+1}$ zeros. As we have shown, A achieves competitive ratio r on advice string ϕ'. Hence, by applying Lemma 1 for $t := r + 1$, the claim of the theorem follows. □

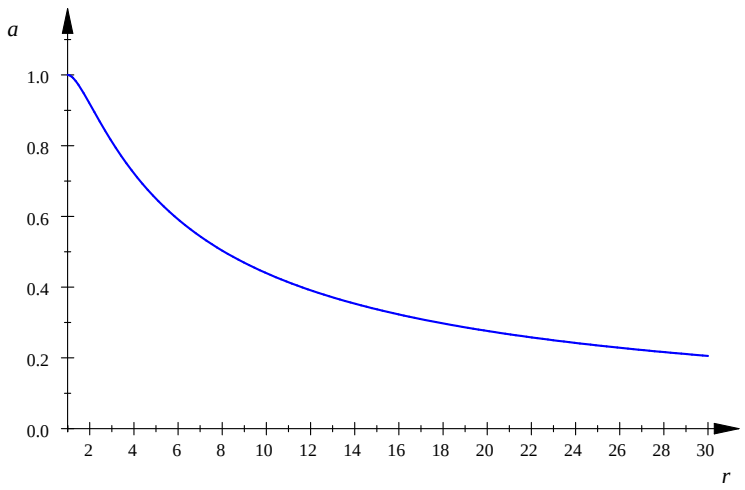


Fig. 2. Number of bits per request ($a = \lim_{n\mapsto\infty} s(n)/n$) sufficient to achieve competitive ratio r

The following corollary expresses the fact that it is possible to obtain a constant competitive ratio with an amortized constant number of bits of advice per request. This number of bits can be arbitrarily close to 0 at the expense of a very high competitive ratio; on the other hand, a competitive ratio arbitrarily close to 1 is reachable with the number of bits per request approaching 1 (see Fig. 2).

Corollary 1 *For each constant $r \geq 1$, there exists an r-competitive algorithm with advice complexity $s(n)$ such that*

$$\lim_{n\to\infty} \frac{s(n)}{n} = \log\left(\frac{r+1}{r^{\frac{r}{r+1}}}\right).$$

Next, we show a lower bound on the advice complexity required to obtain a constant competitive ratio.

Lemma 2 *Consider any paging algorithm A with advice complexity $s(n)$. For any integer n, and any $\gamma \in \left(0, \frac{1}{2}\right)$, there exists an input instance I of length n such that*

$$\frac{C(\mathtt{A}(I))}{C(\mathtt{OPT}(I))} \geq 1 + \frac{\left\lceil \frac{K-1}{2} \right\rceil}{K-1} \cdot \frac{1 + \log(1-\gamma) - \frac{s(n)}{\nu}}{\log\left(\frac{1}{\gamma} - 1\right)},$$

where $\nu := \lfloor n/(2K-2) \rfloor$ and $\mathtt{OPT}(I)$ is the optimal solution of instance I. Furthermore, $C(\mathtt{OPT}(I)) = \nu(K-1)$.

Proof. Consider an algorithm $\mathtt{A}$ with advice complexity $s(n)$, and let $Z = \binom{2K-2}{K-1}$. The overall structure of the proof is as follows. For any ν, we construct a set $\mathcal{I}$ of Z^ν inputs of length n. Obviously, there must be a subset $\mathcal{I}' \subset \mathcal{I}$ of size at least $\frac{Z^\nu}{2^{s(n)}}$ such that the algorithm uses the same advice for all inputs of $\mathcal{I}'$. Then we show how to select an input from $\mathcal{I}'$ on which the competitive ratio cannot be better than claimed by this lemma.

The inputs from $\mathcal{I}$ are organized in a complete Z-ary tree of height ν, where each edge is labeled with a sequence of $2K-2$ requests. Each leaf of the tree represents one input from $\mathcal{I}$ which is obtained by concatenating the request sequences on all edges of the path from the root to this leaf. Let N_h denote the sequence of $K-1$ requests $(hK+1, hK+2, \ldots, hK+K-1)$. Let us denote the root to be on level 1, the sons of the root to be on level 2, etc. Each edge e leading from a vertex v of level h is labeled with the sequence N_h followed by the sequence $S(e)$. The sequences $S(e)$ are defined recursively as follows. For each vertex v, consider a set of K elements $m(v)$; the intuition is that $m(v)$ is the content of the memory of some optimal algorithm processing the given input. Let us define $m(root) = \{1, \ldots, K\}$. Inductively, consider a vertex v of level h with $m(v)$ already defined. The sequences $S(e)$ of the outgoing edges contain $(K-1)$-element subsets of the set $m(v) \cup N_h$ such that they do not contain the element $hK+K-1$ (the ordering within the sequence is not important; to avoid ambiguity, let us assume the sequences $S(e)$ are increasing). For any edge $e = (v, v')$, let us set $m(v') = S(e) \cup \{hK+K-1\}$. Since, by using this procedure, we keep the invariant that $m(v) \cap N_h = \emptyset$, there are exactly $Z = \binom{2K-2}{K-1}$ possible subsets, a unique one for every son of v.

In this way, we obtain a set of instances of length $\nu(2K-2)$. Each of these instances can be extended to length n by repeating the last request $n - \nu(2K-2) = n \bmod (2K-2)$ times.

Now consider any input from $\mathcal{I}$. Since the sequence N_h contains always $K-1$ new pages, there are at least $\nu(K-1)$ page faults made in each algorithm. Moreover, there is an (offline) algorithm that makes no page faults during the processing of $S(e)$.

Now we describe how to select a particular input with large competitive ratio. Consider the algorithm $\mathtt{A}$ with the particular advice for processing the inputs from $\mathcal{I}'$, and denote, for each vertex v, by $m_{\mathtt{A}}(v)$ the contents of the memory of the algorithm. The situation is depicted in Fig. 3: the algorithm has memory $m_{\mathtt{A}}(v)$, and receives the input sequence N_h. Since the algorithm is deterministic, the contents of the memory of the algorithm after processing N_h is the same for all sons of v, and we denote it by $m'_{\mathtt{A}}(v)$. After that there are Z possible ways to extend the input by a sequence $S(e)$. Let

$$\mathit{fault}(S(e)) = |\{x \in S(e)\ ;\ x \notin m'_{\mathtt{A}}(v)\}|,$$

i. e., $\mathit{fault}(S(e))$ is the number of faults made by the algorithm on the sequence of requests $S(e)$. Next, we argue that among the Z sons of v there are at least $Z/2$ with fault at least $(K-1)/2$. In order to show this, let us assign to each set $S(e)$ a complement set $\overline{S(e)}$ of the form

$$\overline{S(e)} = (m(v) \cup N_h) \setminus (S(e) \cup \{Kh-K-1\})\,.$$

Obviously, $\overline{S(e)} = S(e')$ for some other e', and the complementarity relation forms a matching on the Z sons of v (note that Z is even). Consider now a set $S(e)$ such that $\mathit{fault}(S(e)) < (K-1)/2$. We argue that in this case $\mathit{fault}(\overline{S(e)}) > (K-1)/2$ (see Fig. 4).

Let $\mathit{fault}(S(e)) < (K-1)/2$. The memory $m'_{\mathtt{A}}(v)$ contains some elements from $m(v)$, some elements from N_h, and possibly some other elements. Moreover, $m'_{\mathtt{A}}(v)$ contains the element $(Kh+K-1) \in N_h$. The light grey part of $m'_{\mathtt{A}}(v)$ on Fig. 4 is covered by $S(e)$,

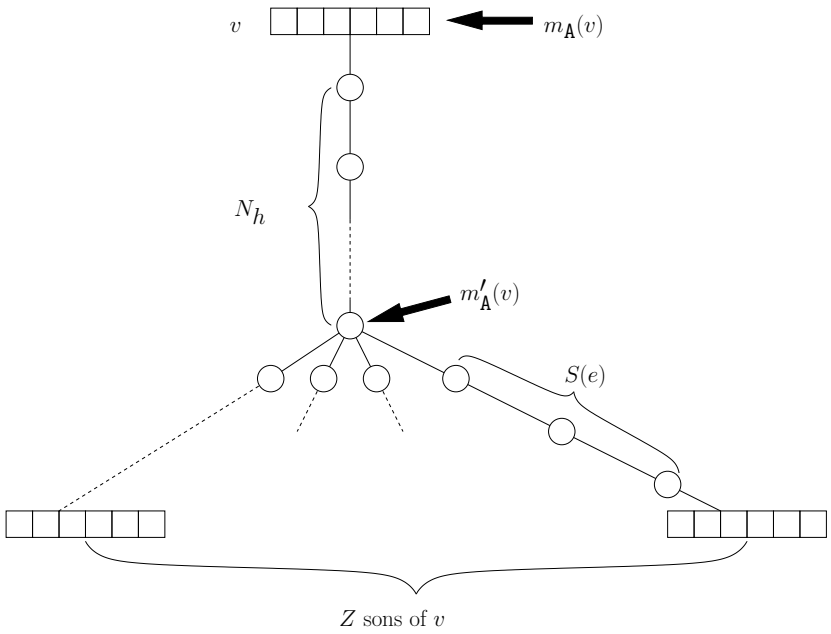


Fig. 3. After processing the input up to some node v, the memory of the algorithm is $m_{\mathtt{A}}(v)$. Since N_h is a common prefix of all the edges from v to its sons, the algorithm has the same contents of memory $m'_{\mathtt{A}}(v)$ after reading the first $|N_h| = K - 1$ requests

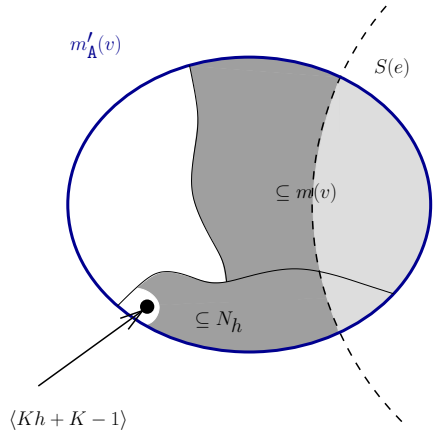


Fig. 4. Illustration of the fact $\mathit{fault}(\overline{S(e)}) > (K-1)/2$

so it contains more than $(K-1)/2$ elements. Hence, the dark grey part contains less than $(K-1)/2$ elements. Furthermore, the dark grey part is equal to $m'_{\mathtt{A}}(v) \cap \overline{S(e)}$, so $\mathit{fault}(\overline{S(e)})$ is greater than $(K-1)/2$.

Now, we select, in each vertex v, exactly $Z/2$ sons with *fault* at least $(K-1)/2$ as *left sons*, the remaining sons are *right*. Let, for a vertex v, $L(v)$ $[R(v)]$ be the number of leaves from $\mathcal{I}'$ in the subtrees induced by left [right] sons. Choose a parameter $\gamma \in (0, 1/2)$, and let us construct an input in the following way starting from the root. For each vertex v, consider two cases:

Case 1: $\gamma(L(v) + R(v)) \leq \min(L(v), R(v))$.
In this case, v is called *balanced*. We select the left son v' with the most elements from $\mathcal{I}'$ in its subtree (this is always non-zero), append the corresponding $S(e)$ to the constructed input, and continue to v'. It is not difficult to check that, for the next iteration,

$$L(v') + R(v') \geq \frac{L(v)}{\frac{Z}{2}} \geq \frac{2\gamma}{Z} \left(L(v) + R(v)\right).$$

Case 2: $\gamma(L(v) + R(v)) > \min(L(v), R(v))$.
In this case, v is called *imbalanced*. Consider the case that $L(v) < \gamma(L(v) + R(v))$, the other case is analogous. Easily, in this case it holds that $R(v) > (1-\gamma)(L(v) + R(v))$, hence, by the pigeonhole principle, some right son of v induces a subtree with at least

$$\frac{(1-\gamma)(L(v) + R(v))}{\frac{Z}{2}}$$

leaves from $\mathcal{I}'$. Similarly as in Case 1, we select the right son v' of v with most elements from $\mathcal{I}'$ in its subtree, append the corresponding $S(e)$ to the constructed input, and continue to v'. This implies

$$L(v') + R(v') \geq \frac{2}{Z}(1-\gamma) \left(L(v) + R(v)\right).$$

Consider the particular input I produced this way. Let w be the number of times a balanced vertex was visited. Since there are ν vertices on the path from the root to any leaf, the procedure starts with $|\mathcal{I}'| = \frac{Z^\nu}{2^{s(n)}}$ leaves, and ends with a single leaf, it follows:

$$1 \geq \frac{Z^\nu}{2^{s(n)}} \left(\frac{2\gamma}{Z}\right)^w \left(\frac{2(1-\gamma)}{Z}\right)^{\nu-w},$$

i. e.,

$$1 \geq 2^{\nu - s(n)} \gamma^w (1-\gamma)^{\nu-w}. \tag{1}$$

The optimal solution for any input from $\mathcal{I}$ is to generate $K-1$ faults in each $S(e)$, i. e., $\nu(K-1)$ faults over the whole input. The algorithm, however, makes at least additional $(K-1)/2$ faults every time a balanced vertex is visited. Hence we get for the ratio:

$$\frac{C(\mathtt{A}(I))}{C(\mathtt{OPT}(I))} \geq \frac{\nu(K-1) + w\left\lceil\frac{K-1}{2}\right\rceil}{\nu(K-1)} = 1 + \frac{w}{\nu} \cdot \frac{\left\lceil\frac{K-1}{2}\right\rceil}{K-1}.$$

From (1), we get

$$\frac{w}{\nu} \geq \frac{1 + \log(1-\gamma) - \frac{s(n)}{\nu}}{\log\left(\frac{1}{\gamma} - 1\right)},$$

which gives the result. □

Using the previous lemma, we can express the minimal number of advice bits per request required to obtain an r-competitive algorithm. The following theorem states this result.

Theorem 2 *Let r be any constant such that $1 \leq r \leq 1.25$. For any paging algorithm $\mathtt{A}$ with advice complexity $s(n)$ and competitive ratio r it holds that*

$$\frac{s(n)}{n} \geq \frac{1}{2K-2}\left[1 + \log(3-2r) - (2r-2)\log\left(\frac{1}{2r-2} - 1\right)\right] - \mathcal{O}\left(\frac{1}{n}\right).$$

The constant of the $\mathcal{O}(1/n)$ depends on K and the parameters r, α of $\mathtt{A}$ (as used in Definition 1).

Proof. For any n, Lemma 2 guarantees existence of an input instance I such that

$$\frac{C(\mathtt{A}(I))}{C(\mathtt{OPT}(I))} \geq 1 + \frac{\left\lceil \frac{K-1}{2} \right\rceil}{K-1} \cdot \frac{1 + \log(1-\gamma) - \frac{s(n)}{\nu}}{\log\left(\frac{1}{\gamma} - 1\right)} \geq 1 + \frac{1}{2} \cdot \frac{1 + \log(1-\gamma) - \frac{s(n)}{\nu}}{\log\left(\frac{1}{\gamma} - 1\right)} \tag{2}$$

holds for every $\gamma \in (0, 1/2)$, where $\nu := \lfloor n/(2K-2) \rfloor$, $\mathtt{OPT}(I)$ is the optimal solution of instance I, and $C(\mathtt{OPT}(I)) = \nu(K-1)$. Let $q := C(\mathtt{A}(I))/C(\mathtt{OPT}(I))$. Equation (2) then implies:

$$\begin{aligned} s(n) &\geq \nu \left(1 + \log(1-\gamma) - \log\left(\frac{1}{\gamma} - 1\right)(2q-2)\right) \\ &\geq \left(\frac{n}{2K-2} - 1\right)\left(1 + \log(1-\gamma) - \log\left(\frac{1}{\gamma} - 1\right)(2q-2)\right), \end{aligned}$$

and thus

$$\frac{s(n)}{n} \geq \frac{1}{2K-2}\left(1 + \log(1-\gamma) - \log\left(\frac{1}{\gamma} - 1\right)(2q-2)\right) - \mathcal{O}\left(\frac{1}{n}\right). \tag{3}$$

Since $\mathtt{A}$ is r-competitive, we have that $C(\mathtt{A}(I)) \leq rC(\mathtt{OPT}(I)) + \alpha$, which yields

$$q = \frac{C(\mathtt{A}(I))}{C(\mathtt{OPT}(I))} \leq r + \frac{\alpha}{C(\mathtt{OPT}(I))} = r + \mathcal{O}\left(\frac{1}{n}\right). \tag{4}$$

Substituting (4) into (3) we have

$$\frac{s(n)}{n} \geq \frac{1}{2K-2}\left(1 + \log(1-\gamma) - \log\left(\frac{1}{\gamma} - 1\right)(2r-2)\right) - \mathcal{O}\left(\frac{1}{n}\right). \tag{5}$$

Since we can choose any $\gamma \in (0, 1/2)$, we need to maximize

$$X(\gamma) := 1 + \log(1-\gamma) - \log\left(\frac{1}{\gamma} - 1\right)(2r-2).$$

It is straightforward to check that $X'(\gamma) = 0$ holds for $\gamma = 2r-2$, so $X(\gamma)$ is maximal in this case. Hence, we have

$$\frac{s(n)}{n} \geq \frac{1}{2K-2}\left(1 + \log(3-2r) - \log\left(\frac{1}{2r-2} - 1\right)(2r-2)\right) - \mathcal{O}\left(\frac{1}{n}\right).$$

□

The previous theorem proves that an amortized constant number of advice bits per request is necessary to obtain algorithms with competitive ratio better than 1.25, see Fig. 5.

To obtain an optimal algorithm, large advice is necessary. The following theorem proves a lower bound on the advice complexity that converges to 1 bit per request with growing cache size K. Hence, the upper bound of Fact 2 is tight for large K.

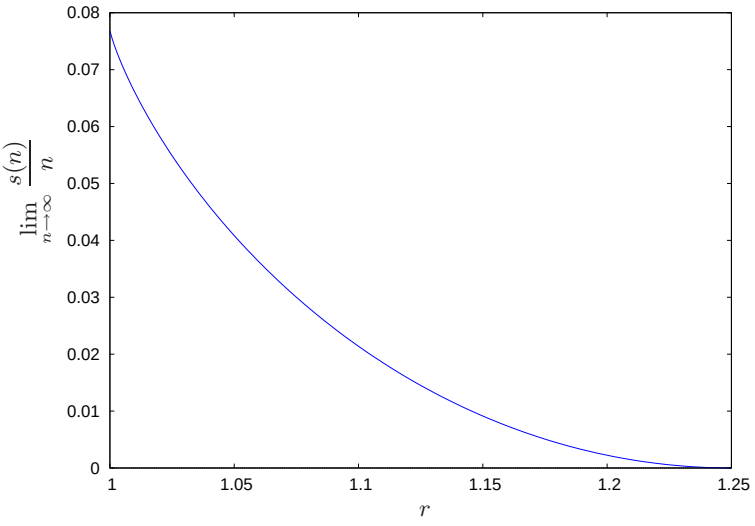


Fig. 5. Number of bits per request ($\lim_{n\mapsto\infty} s(n)/n$) necessary to achieve competitive ratio r, values for $K = 7$

Theorem 3 *For any optimal paging algorithm* A *with advice complexity* $s(n)$*,*

$$\frac{s(n)}{n} \geq \frac{1}{2K-2}\log\binom{2K-2}{K-1} - \mathcal{O}\left(\frac{1}{n}\right) \geq 1 - \frac{\log(K-1)+C}{4(K-1)} - \mathcal{O}\left(\frac{1}{n}\right)$$

for some constant C *not depending on* K*. The constant of* $\mathcal{O}(1/n)$ *depends on* K*.*

Proof. Consider the set of inputs $\mathcal{I}$ as introduced in the proof of Lemma 2 for some n; there are Z^ν of them, where $Z = \binom{2K-2}{K-1}$, and $\nu = \lfloor \frac{n}{2K-2} \rfloor$.

It is not difficult to see that A has to use different advice on any two inputs of $\mathcal{I}$. Otherwise, A makes the same decision on edges outgoing from the lowest common ancestor of some inputs $I_1, I_2 \in \mathcal{I}$, and makes at least one more page fault than the optimal algorithm on these edges in at least one of these inputs, thus A cannot be optimal.

Hence, the advice complexity of A has to be at least $\log(Z^\nu) = \nu \log(Z)$ bits, which proves the first claim of the theorem:

$$\begin{aligned} s(n) &\geq \nu \log(Z) = \left\lfloor \frac{n}{2K-2} \right\rfloor \log\binom{2K-2}{K-1} \\ &\geq \left(\frac{n}{2K-2} - 1\right)\log\binom{2K-2}{K-1} = \frac{n}{2K-2}\log\binom{2K-2}{K-1} - \mathcal{O}(1). \end{aligned}$$

This implies

$$\frac{s(n)}{n} \geq \frac{1}{2K-2}\log\binom{2K-2}{K-1} - \mathcal{O}\left(\frac{1}{n}\right). \tag{6}$$

Using the Stirling formula, we have

$$\binom{2t}{t} = \frac{(2t)!}{(t!)^2} = \frac{\sqrt{2\pi 2t}\left(\frac{2t}{e}\right)^{2t}\left(1+\mathcal{O}\left(\frac{1}{t}\right)\right)}{\left(\sqrt{2\pi t}\left(\frac{t}{e}\right)^{t}\left(1+\mathcal{O}\left(\frac{1}{t}\right)\right)\right)^2} = 2^{2t}\cdot\frac{D}{\sqrt{t}} \tag{7}$$

for some constant D not depending on K. Substituting $t := K - 1$ into (7) and (6), we complete the proof:

$$\begin{aligned}\frac{s(n)}{n} &\geq \frac{1}{2K-2}\log\left(2^{2(K-1)}\frac{D}{\sqrt{K-1}}\right) - \mathcal{O}\left(\frac{1}{n}\right)\\ &= \frac{2(K-1)+\log(D)-\frac{1}{2}\log(K-1)}{2K-2} - \mathcal{O}\left(\frac{1}{n}\right)\\ &= 1 - \frac{-2\log(D)+\log(K-1)}{4K-4} - \mathcal{O}\left(\frac{1}{n}\right).\end{aligned}$$

□

3.3 Small Advice

In this section, we analyze the case where only very small advice is available to the algorithm. More precisely, we focus on the case where the advice complexity $s(n)$ is constant (independent of n), i. e., where the algorithm is allowed to use only constant number of advice bits for the whole input. We show that, even in this very restricted case, the competitive ratio can be significantly improved with respect to the best deterministic algorithm. We start with the upper bounds on advice complexity, and complement them later with a lower bound.

The next theorem focusses on the special case of Paging with only $K+1$ logical memory pages, i. e., there is always exactly one logical page missing in the physical memory.

Theorem 4 *Consider the class of inputs with $K+1$ possible pages, where K is the size of the buffer, and let $c < K$ be a power of 2. There is an algorithm with advice complexity $s(n) = \log c$ for* Paging *on this class of inputs with competitive ratio*

$$r \leq \frac{3}{2}\log c + \frac{K-1}{c}.$$

Proof. We construct c deterministic algorithms $\mathtt{A}_1, \ldots, \mathtt{A}_c$, and argue that, for each input, some of them does not make too many faults. The advice is then the number of the algorithm that should be used to process this input.

The algorithms will be marking algorithms according to Definition 5, and as such they possess a natural concept of a *phase*: a phase starts in step 4 of Definition 5, when the algorithm unmarks all pages, i. e., after K distinct requests have arrived since the start of the previous phase. Note that, for a given input, the sets of marked pages in any step (and hence also the beginnings of phases) are identical for all marking algorithms.

By definition, we assume that all algorithms start with the pages $\{1, 2, \ldots, K\}$ in memory, and all of them are marked, so the first request is the page $K+1$ and it starts a new phase. Since the contents of the memory are identical for all algorithms at the beginning of any phase, it is sufficient to describe the processing of a single (first) phase; all other phases differ only by having different sets of starting pages. Furthermore, since any request for a marked page does not cause a page fault in any algorithm, we can ignore such requests, so that each phase consists of exactly K requests $r_0, r_1, \ldots, r_K$. After processing requests $r_0, \ldots, r_t$, the memory of any algorithm $\mathtt{A}_i$ contains marked pages $r_0, \ldots, r_t$, and some $K-t-1$ unmarked pages. Now we describe the behavior of each of the algorithms on requests $r_0, r_1, \ldots, r_K$.

The request $r_0 = K+1$ causes a page fault in all algorithms; algorithm $\mathtt{A}_i$ replaces page i by $K+1$. The remaining $K-1$ requests are processed in rounds. Round 0 starts by processing

request r_1, and lasts $K-c-1$ steps. Any other round t for $t>0$ starts by processing request $r_{K-\frac{c}{2^t-1}}$ and lasts $\frac{c}{2^t}$ steps. In each round t, the algorithms are partitioned into groups of size 2^t, and each group G_j has an *assigned page* p_j, such that for any two groups G_j, $G_{j'}$, $j \neq j'$, the assigned pages differ, i. e., $p_j \neq p_{j'}$. In round 0, each algorithm forms a group of its own, and algorithm $\mathtt{A}_j$ has the assigned page $p_j = j$. At the beginning of the round $t > 0$, groups G_{2j} and G_{2j+1} are merged into a single group with assigned page p_{2j}. Let r be any request that causes a page fault in some algorithm $\mathtt{A}_i$ belonging to group G_j during round t. Let p be the smallest unmarked page that is not assigned to any group (note that during round t there are more than $c/2^t$ unmarked pages but only $c/2^t$ groups). If p_j is in $\mathtt{A}_i$'s memory, $\mathtt{A}_i$ replaces p_j by r. Otherwise, if p_j is missing in $\mathtt{A}_i$'s memory, clearly p must be in $\mathtt{A}_i$'s memory (since there is only one page missing); $\mathtt{A}_i$ then replaces p by $r = p_j$, and changes the assigned page of its group G_j to p (note that if more algorithms from the same group perform this operation, they assign the same number p to the group). Note that this operation never violates the invariant that different pages are assigned to any two groups: It is not difficult to check that there can be at most one group changing its assigned page in one request (otherwise the invariant did not hold before that particular request), and the newly assigned page p was not assigned to any group by the choice of p.

Now let us bound from above the overall number of page faults occurred in all algorithms during one phase. The first request causes a page fault in every algorithm, so the contribution is c faults. During the round 0, it is easy to see that, for every algorithm $\mathtt{A}_j$ the assigned page p_j is missing in $\mathtt{A}_j$'s memory. Hence, each request can cause a page fault in at most one algorithm. Since the length of the round is $K-c-1$, the overall contribution of this round is at most $K-c-1$ faults.

Consider now an arbitrary round $t > 0$. At the beginning, we charge a credit of one page fault to every algorithm from the merged groups G_{2j+1}, this is a contribution of $c/2$ page faults that will be accounted for later during the round. First note that, if an algorithm is missing p_j in the memory, it remains missing p_j in memory until the end of the round. Moreover, the only way an algorithm may be missing a page different from p_j in the memory is to be in the group G_{2j+1} during the merging.

Consider a request r that causes a page fault to an algorithm $\mathtt{A}_i$ from group G_j. Let us distinguish two cases. If $p_j = r$ is missing in $\mathtt{A}_i$'s memory: since the assigned pages for different groups are different, there are at most 2^t such algorithms accounting for at most 2^t page faults during this particular request. If p_j is in $\mathtt{A}_i$'s memory, the credit charged to $\mathtt{A}_i$ during the last time it was merging from an odd group is accounted now; after this step, $\mathtt{A}_i$ is missing p_j in memory.

Since round $t > 0$ has $c/2^t$ steps, there are at most

$$\frac{c}{2} + 2^t \frac{c}{2^t} = \frac{3}{2}c$$

page faults during this round. Hence, the overall number of page faults during one phase is at most

$$K - 1 + \frac{3}{2} c \log c.$$

Consider an input X consisting of f phases. Then the overall number of page faults made by all algorithms is at most $f \cdot \left(K - 1 + \frac{3}{2} c \log c\right)$. Since there are c algorithms, there must be one of them that makes at most

$$f \cdot \left(\frac{K-1}{c} + \frac{3}{2} \log c \right)$$

faults on X.

On the other hand, any algorithm (not necessarily a marking one) makes at least f page faults: consider any two consecutive phases $i-1$, i, and assume that there is no page fault done while processing requests $2, \ldots, K$ of phase $i-1$. Since there are K distinct requests in phase $i-1$, in this case the memory must contain all these pages at the end of the phase. Hence the first request from the next phase i generates a fault. Furthermore, it is easy to see that the first request of the first phase generates a fault, too. So, there are at least f page faults in total. □

In the next theorem, we generalize this result for an arbitrary number of logical pages.

Theorem 5 *Consider the class of inputs with a buffer of size K, and let $c < K$ be a power of 2. There is an algorithm with oracle size $\log c$ for* Paging *on this class of inputs with competitive ratio*

$$r \leq 3 \log c + \frac{2(K+1)}{c} + 1.$$

Proof. Similarly as in Theorem 4, we construct c deterministic marking algorithms $\mathtt{A}_1, \ldots, \mathtt{A}_c$, and argue that, for each input, one of them does not make too many page faults. Again, we use the concept of a phase, and assume that the first request starts a new phase.

Analogously to Theorem 4, it is sufficient to describe the processing of a single (first) phase consisting of exactly K requests $r_0, r_1, \ldots, r_{K-1}$. Each request r_t can be a request for some unmarked *starting* page $r_t \leq K$, i. e., a page that was in the memory at the beginning of the phase, or a request for some *new* page $r_t > K$. After processing requests $r_0, \ldots, r_{t-1}$, the memory of any algorithm $\mathtt{A}_j$ contains marked pages $r_0, \ldots, r_{t-1}$, and some $K-t$ unmarked pages. Now we describe the behavior of each of the algorithms on requests $r_0, r_1, \ldots, r_K$.

The K requests are processed in $\log c + 1$ rounds. Round 0 starts by processing request r_0, and lasts $K - c + 1$ steps. Any other round u for $u > 0$ starts by processing request $r_{K - \frac{c}{2^{u-1}} + 1}$ and lasts $\frac{c}{2^u}$ steps. In each round u, the algorithms are partitioned into groups $G_0, \ldots G_{c/2^u - 1}$ of size 2^u.

Consider request r_t that is processed in round u. Let m_t be the number of requests of new pages among $r_0, \ldots r_{t-1}$. We maintain an ordering of all unmarked starting pages. Since there are $K + m_t - t$ such pages when request r_t arrives, we denote them as $s_0, \ldots, s_{K+m_t-t-1}$. When request r_0 arrives, we can start with an arbitrary order of the K starting pages. We say that group G_i has an *assigned set of pages* $p_i^{(t)} = \{s_i, \ldots, s_{i+m_t-1}\}$, i. e., a set of m_t consecutive unmarked starting pages. It is easy to check that every unmarked starting page is assigned to at most m_t groups, and that the last

$$(K + m_t - t - 1) - (c/2^u - 1 + m_t - 1) = K - t - c/2^u + 1 \geq 1$$

unmarked starting pages are not assigned to any group.

In round 0, each algorithm forms a group of its own. At the beginning of a round $u > 0$, groups G_i and $G_{i+c/2^u}$ are merged to a single group G_i. Now we show how to process request r_t. At first, we describe how to modify the ordered sequence of unmarked starting pages: If r_t is a request for a new page, no change is necessary. In this case, every group is assigned one more page after processing r_t. If r_t is a request for a starting page, we need to remove the page r_t from the sequence. To do so, we replace the page by any starting page that is not assigned to any group. As proved above, the last page of the sequence is never assigned

to any group, hence such page always exists. In this case, assignments of at most m_t groups were changed, and the only change was replacement of page r_t by some other page.

Consider any algorithm $\mathtt{A}_j$ belonging to group G_i such that r_t causes a page fault in $\mathtt{A}_j$. To process the request, $\mathtt{A}_j$ replaces any unmarked starting page assigned to G_i by r_t in its memory. We need to show that such page always exists. When r_t arrives, $\mathtt{A}_j$ has $K-t$ unmarked starting pages in memory. After modifying the ordered sequence of unmarked starting pages, there are m_{t+1} pages assigned to G_i, and $K+m_{t+1}-(t+1)$ unmarked starting pages in total. Hence, there are only $K-t-1$ unmarked starting pages not assigned to G_i, so the memory of $\mathtt{A}_j$ contains some page assigned to G_i.

Now let us bound from above the overall number of page faults occurring in all algorithms during one phase. To do so, we use the following accounting argument. When assigning unmarked starting pages to groups, we distribute coins on memory cells of all algorithms such that the following invariant holds: *Each memory cell of an algorithm $\mathtt{A}_j$ belonging to group G_i that contains a page assigned to G_i has a coin.* It is easy to see that every page fault of any algorithm consumes one coin. Hence, it is sufficient to count how many coins have to be distributed.

No coins are necessary at the beginning of round 0. At the beginning of any round $u > 0$ starting by request r_t, groups G_i and $G_{i+c/2^u}$ are merged together. Obviously, assigned pages of algorithms in G_i do not change. Any algorithm $\mathtt{A}_j$ in $G_{i+c/2^u}$ gets at most m_t newly assigned pages after the merge, so $\mathtt{A}_j$ requires at most m_t extra coins. Since there are $c/2$ affected algorithms, $m_t c/2 \leq m_K c/2$ extra coins are sufficient, which sum up to $m_K \frac{c}{2}(\log c + 1)$ for all rounds.

Consider any request r_t for a new page. One additional page is assigned to every group, hence at most one extra coin is needed in every algorithm. Since there are m_K requests for new pages, such requests can be covered by $m_K c$ extra coins.

Finally, consider any request r_t for a starting page in round $u \geq 0$. To process the request, m_t groups are assigned an additional page. Since each group has 2^u algorithms, we need $m_t 2^u \leq m_K 2^u$ extra coins to cope with that. For $u > 0$, there are at most $c/2^u$ such requests in round u, so we need $m_K c$ coins for any round $u > 0$. For $u = 0$, we need $m_K(K - c + 1)$ coins. Hence, $m_K c \log c + m_K(K - c + 1)$ coins are sufficient for all rounds. Adding everything together, $m_K \frac{c}{2}(3 \log c + 1) + m_K(K + 1)$ coins are sufficient in total.

Consider an input X with m requests for new pages. Then the overall number of page faults made by all algorithms is at most $m\frac{c}{2}(3 \log c + 1) + m(K + 1)$. Since there are c algorithms, there must be one of them that makes at most

$$m\left(\frac{K+1}{c} + \frac{3}{2}\log c + \frac{1}{2}\right)$$

faults on X.

On the other hand, any algorithm makes at least $m/2$ page faults (see Theorem 4.3 in [2]). □

The presented results show that even very small advice can be used efficiently. Furthermore, $\log K$ bits of advice (i. e., making c equal to the largest power of 2 smaller than K) can be used to achieve the competitive ratio $3 \log K + \mathcal{O}(1)$ which is asymptotically equal (albeit the constant is worse) to the best possible ratio $H_K = \sum_{i=1}^{K} \frac{1}{i}$ of a randomized algorithm without advice.

Next, we complement the result of Theorem 5 by presenting a lower bound on the competitive ratio for paging algorithms with constant advice complexity.

Theorem 6 *Consider the class of inputs with $K+1$ possible pages, where K is the size of the buffer, and let c be a power of 2. Any deterministic algorithm* A *with advice complexity $s(n) = \log c$ for* Paging *on this class of inputs has a competitive ratio of at least K/c.*

Proof. Consider all inputs of length n ordered into a K-ary tree: each vertex of the tree has K sons representing K possible pages that could arrive in the next request (consecutive sequences of the same page are ignored). The leaves of the tree correspond to particular inputs, and every vertex v of a tree corresponds to a prefix of some inputs (exactly those which correspond to the leaves of the subtree induced by v). With $\log c$ bits of advice, there are c possible advice strings, and each input has to be processed using one of the advices. Let us color the leaves with c colors according to which advice is used to process the input. Let $h := \lfloor n/c \rfloor$.

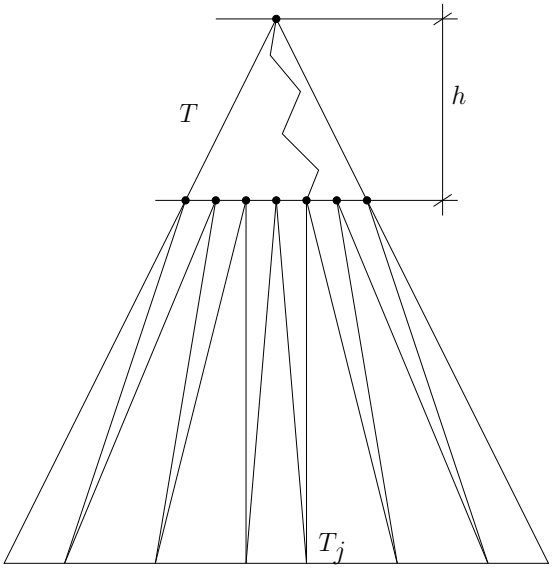


Fig. 6. Cutting of the input-tree. If there is a subtree T_j with some color missing, the induction is used on T_j, otherwise there is a color that covers the whole tree T.

We show, by induction on i, that, if there is a vertex v inducing a subtree T_v of height at least $h \cdot i$ such that all leaves of T_v are colored by at most i colors, there is an input where the algorithm makes at least h faults. The claim of the theorem follows immediately from the case $i = c$.

Consider a subtree T_v of height at least $h \cdot i$ with leaves colored with i colors. If $i = 1$, then all inputs of T_v are processed using the same advice. Hence, we have an ordinary deterministic algorithm that has to process all inputs of T_v. Obviously, there is an input where the algorithm makes a page fault in every step in T_v, so it makes at least h faults.

Let $i > 1$, and cut the tree T_v after h levels, so that we have a tree T of height h, and there is a subtree T_j of height $h(i-1)$ rooted at each leaf of T. Let us distinguish two cases. If there is a subtree T_j with leaves colored by $i-1$ colors we can use induction to argue that there are at least h faults on some input belonging to T_j. So suppose that there is a color a that is used in all subtrees. Select a leaf colored by a from every subtree; these inputs are processed with the same advice. However, the requests of these inputs belonging to T cover all

possible request sequences of length h. Hence, we can argue that there is one of them where the algorithm makes h faults.

To conclude the proof, note that the optimal algorithm makes at most n/K faults: consider a page fault caused by a page p. Since there are only $K+1$ possible pages, and all of them except for p are in the memory, there must be a page that will not be requested during the next $K-1$ requests. Hence, the optimal algorithm makes at most one page fault per K requests.

For the ratio r of the algorithm $\mathtt{A}$ and any input instance I, it holds that $C(\mathtt{A}(I)) \leq r \cdot C(\textsc{Opt}(I)) + \alpha$, where $\textsc{Opt}$ is an optimal offline algorithm for Paging. Since we have proven the existence of an input instance I where $C(\textsc{Opt}(I)) \leq n/K$ and $C(\mathtt{A}(I)) \geq \lfloor n/c \rfloor$, we obtain that $r \geq K/c$. □

4 Disjoint Path Allocation

In this section, we consider a special type of network topology where the entities of a network are connected by one line (i. e., a bus network). For $L+1$ entities, the network is a path P of length L. All connections in P have a capacity of 1. For the *disjoint path allocation problem* (as described in [2]), additionally a set of subpaths of P is given. Each subpath $(v_i, v_{i+1}, \ldots, v_j)$ is a so-called *call request*, i. e., a request to establish a permanent connection between the two endpoints v_i and v_j. If such a request is satisfied, no inner entity of the path is able to be part of any other call. Therefore, a *disjoint path allocation* is simply a set of edge-disjoint subpaths of P.

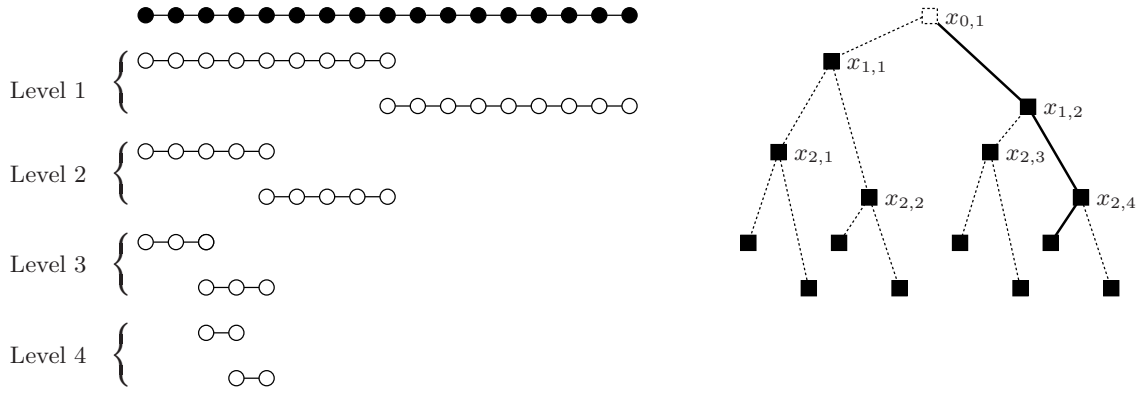
Definition 6 (Disjoint Path Allocation Problem). Given a path $P = (V, E)$, where $V = \{v_0, v_1, \ldots, v_L\}$ is a set of entities, $|V| = L+1$, and a set $\mathcal{P}$ of subpaths of P where $|\mathcal{P}| = n$, DisPathAlloc is the problem of finding a maximum set $\mathcal{P}' \subseteq \mathcal{P}$ of edge-disjoint subpaths of P.

In this paper, we deal with an environment in which the subpaths $P_1, P_2, \ldots, P_n \in \mathcal{P}$ arrive in an online fashion. We assume that L is known to the online algorithm in advance, but this is not the case for n. For the ease of presentation, let (v_i, v_j) denote the path $(v_i, v_{i+1}, \ldots, v_j)$. Since DisPathAlloc is a maximization problem, an algorithm $\mathtt{A}$ solving it is c-competitive if $C(\textsc{Opt}(I)) \leq c \cdot C(\mathtt{A}(I)) + \alpha$ for some constant α and every input I (cf. Definition 1). For a maximization problem, we use terms *cost* and *gain* interchangeably.

By some simple observations, it is clear that any deterministic online algorithm $\mathtt{A}$ is no better than L-competitive: An adversary controlling $\mathcal{P}$ simply sends $P_1 = (v_0, v_L)$ as the first part of the input. Discarding it results in the adversary not sending any other subpath and therefore $\mathtt{A}$ has to always satisfy the first request. The adversary now sends the consecutive requests $P_2 = (v_0, v_1), P_3 = (v_1, v_2), \ldots, P_{L+1} = (v_{L-1}, v_L)$, none of which can be satisfied. Obviously, an optimal offline algorithm would discard P_1 in this case and satisfy all L requests P_2 to P_{L+1}. Note that there are exactly $L+1 = n$ requests for this input and $\mathtt{A}$ is therefore also no better than $(n-1)$-competitive.

This lower bound is tight: Consider a simple greedy algorithm $\mathtt{G}$ which always satisfies the first request given and then any other request possible.

Lemma 3 *The algorithm* $\mathtt{G}$ *is both* $(n-1)$*-competitive and* L*-competitive.*



(a) A sample input from the class $\mathcal{I}_3$

(b) $\mathcal{T}$ and an optimal solution represented by a path of length h

Fig. 7. Example input instance for DisPathAlloc

Proof. Since G always satisfies at least one request, we merely need to consider cases in which an optimal offline algorithm gains strictly more than $n-1$ requests to prove $(n-1)$-competitiveness. However, this is only the case if no subpath of the input intersects with any other subpath, but in this case, G is optimal.

To prove that G is L-competitive, note that G satisfies at least one request, but no solution can satisfy more than L requests. □

Before we continue with the investigation of lower and upper bounds on the number of advice bits needed by online algorithms to yield a specific competitive ratio, we have to introduce a special class of input instances which are needed in the following proofs.

4.1 The Class $\mathcal{I}$ of Inputs

For some of the following proofs in this section, we consider input instances as depicted in Fig. 7 (a) for any even n. For every h, we define the class $\mathcal{I}_h$ (and $\mathcal{I} = \bigcup_{h\in\mathbb{N}} \mathcal{I}_h$) in the following way. Every element of $\mathcal{I}_h$ consists of $h+1$ so-called *levels*. Level 1 consists of two disjoint consecutive requests, splitting the line network into two parts of the same size. After that, two disjoint consecutive requests on level $i+1$ do the same with one of the intervals from level i. This is iterated until two requests of size 1 appear on level $h+1$. Fig. 7 (a), for instance, shows an input from $\mathcal{I}_3$. It is obvious that any optimal algorithm satisfies exactly one interval on the first h levels, allowing it to satisfy both on level $h+1$. In this example, an optimal strategy is to satisfy the second request on level 1 and 2, the first request on level 3, and correspondingly being able to satisfy the two requests on level 4. Note that, if an algorithm diverges from this strategy, it is not able to satisfy any more requests for this instance afterwards.

Clearly, we may represent an optimal solution for any input sequence as described above by a path from the root to a leaf in a complete binary tree $\mathcal{T}$ of height h with its root on a notional level 0 (see Fig. 7 (b)). The 2^h leaves of $\mathcal{T}$ represent the 2^h different inputs of the class $\mathcal{I}_h$. Let $\mathcal{OPT}$ denote a path corresponding to some input instance $\in \mathcal{I}_h$. We may say that an optimal algorithm Opt makes moves *according to this path*. For an arbitrary online algorithm A that satisfies one request on level i, we say that A makes the *right* decision on this level if

it also acts according to $\mathcal{OPT}$. However, if $\mathtt{A}$ satisfies one interval not according to this path $\mathcal{OPT}$ or satisfies both requests, we say that $\mathtt{A}$ makes a *wrong* decision on level i. In this case, we say that $\mathtt{A}$ is *out* (after level i). Furthermore, for every i and an input instance $I_h \in \mathcal{I}_h$, let $C_i(\mathtt{A}(I_h))$ denote the overall number of requests satisfied by $\mathtt{A}$ up to level i. If $\mathtt{A}$ is *out* after level i, this means that, for every $j \geq i$, $C_j(\mathtt{A}(I_h)) = C_i(\mathtt{A}(I_h))$ (and obviously, therefore, $C(\mathtt{A}(I_h)) = C_i(\mathtt{A}(I_h))$). Let $right(\mathtt{A})$ [$wrong(\mathtt{A})$] denote the set of time steps in which $\mathtt{A}$ makes the right [wrong] decision. Since making the wrong decision can only happen once (because the algorithm is out afterwards), the overall gain of $\mathtt{A}$ is

$$C(\mathtt{A}(I_h)) \leq |right(\mathtt{A})| + 2 \cdot |wrong(\mathtt{A})| \leq |right(\mathtt{A})| + 2 \tag{8}$$

which directly implies that, for an optimal algorithm $\textsc{Opt}$, $C(\textsc{Opt}(I_h)) = h + 2$.

4.2 Advice Complexity Bounds with respect to the Number of Requests

First, we show that DisPathAlloc is a hard online problem even for randomized algorithms. For a randomized algorithm $\mathtt{R}$, let $\mathcal{R}$ denote $\mathtt{R}$'s output and $C(\mathcal{R})$ the overall number of requests satisfied by $\mathcal{R}$. We prove the following theorem stating that any randomized algorithm is at least $(n/6 - \mathcal{O}(1))$-competitive.

Theorem 7 *For every randomized online algorithm* $\mathtt{R}$ *for* DisPathAlloc*, there exists an input such that* $E[C(\mathcal{R})] \leq 3$ *whereas an optimal solution satisfies at least* $n/2$ *requests.*

Proof. Consider the class of inputs $\mathcal{I}$ as shown in Fig. 7 (a). For any randomized algorithm, we construct a hard instance from $\mathcal{I}_h$. Any of these instances containing $n = 2(h+1)$ requests can be described by a string $s = s_1 \dots s_h \in \{L, R\}^h$ where $s_i = L$ [$s_i = R$] means that the optimal algorithm satisfies the left [right] interval on level i, i. e., the intervals corresponding to the requests on level $i+1$ (and thus on all subsequent levels) are sub-intervals of the right [left] interval on level i.

Recall that an optimal offline algorithm can choose exactly one request on levels $1, \dots, h$ and both requests on level $h+1$, i. e., $h + 2 \geq n/2$ requests in total.

Let $\mathtt{R}$ be an arbitrary randomized algorithm for DisPathAlloc which, on inputs from $\mathcal{I}_h$, chooses at most one request from each level. The remaining case of algorithms which are also allowed to take both requests from one level will be dealt with later.

Let Ω denote the sample space, i. e., the set of states in which $\mathtt{R}$ may be at a specific time step. Each of these states contains the complete information about the input instance, the current level and the random decisions made up to this time step. For any state $\omega \in \Omega$, let $p_L(\omega)$ [$p_R(\omega)$] denote the probability with which $\mathtt{R}$ will choose the left [right] request on the current level, if it is in state ω. Analogously, let $p_0(\omega)$ denote the probability of $\mathtt{R}$ choosing no request on the current level. We partition the state space into two parts Ω_{alive} and Ω_{out}, where Ω_{alive} contains all states in which $\mathtt{R}$ may still be able to choose more requests and Ω_{out} is the set of states in which $\mathtt{R}$ is out.

We define the following random variables on Ω: Let $V\colon \Omega \to \mathbb{N}$, where $V(\omega)$ is the number of already chosen intervals in state $\omega \in \Omega$. We call $V(\omega)$ the *gain* of $\mathtt{R}$ in state ω. Furthermore, we define $P_L, P_R\colon \Omega \to [0,1]_{\mathbb{R}}$ as

$$P_L(\omega) = \begin{cases} 0, & \text{if } \omega \in \Omega_{out} \\ p_L(\omega), & \text{if } \omega \in \Omega_{alive} \end{cases}, P_R(\omega) = \begin{cases} 0, & \text{if } \omega \in \Omega_{out} \\ p_R(\omega), & \text{if } \omega \in \Omega_{alive} \end{cases},$$

and $P_0 \colon \Omega \to [0,1]_{\mathbb{R}}$ as

$$P_0(\omega) = \begin{cases} 0, & \text{if } \omega \in \Omega_{out} \\ p_0(\omega), & \text{if } \omega \in \Omega_{alive} \end{cases}.$$

Then, P_L is a random variable describing the probability that, in the current state, R is still alive and chooses the left request, and P_R and P_0 are defined analogously.

Every input instance s (and also every prefix of an input instance) induces a probability distribution $\mathcal{P}_s$ over Ω. By $E^s[X]$, we denote the expected value of the random variable X in the probability space $(\Omega, \mathcal{P}_s)$.

We consider the following expected values: $E^s[V]$ is the expected number of requests chosen by R on input s. Moreover, $E^s[P_L]$ is the expected probability of R being alive after reading s and taking the left request on the last level, and $E^s[P_R]$ and $E^s[P_0]$ are defined analogously.

Observe that $E^s[P_L + P_R + P_0]$ describes the expected probability of R being alive after reading the input s.

We now inductively define a hard input instance for R. For the empty string λ, we obviously have $E^\lambda[V] = 0$ and $E^\lambda[P_L + P_R + P_0] = 1$. Assume now that we have already chosen a prefix s of our input string. Without loss of generality, we may assume that

$$E^s[P_L] \leq E^s[P_R]. \tag{9}$$

Then, we extend the input string to $s' = s \cdot L$. In the probability space $(\Omega, \mathcal{P}_{sL})$, we have

$$E^{sL}[V] = E^s[V] + (E^s[P_L] + E^s[P_R]) \cdot 1 \leq E^s[V] + 2 \cdot E^s[P_R]. \tag{10}$$

Here, the first equality holds since the expected number of chosen intervals increases by one exactly in those cases where R is still alive after having processed the prefix s and additionally chooses one of the requests on the last level, whereas the inequality directly follows from assumption (9).

On the other hand, the expected probability of R to still be alive after reading the extended input can be computed as

$$E^{sL}[P_L + P_R + P_0] = E^s[P_0 + P_L] = E^s[P_L + P_R + P_0] - E^s[P_R]. \tag{11}$$

Here, the first equality follows from the fact that R is out if it chooses the right request after reading s, but stays alive if it chooses the left or none of the two requests. The second equality simply follows from linearity of expectation.

According to Equations (10) and (11), for every level, the expected additional gain of R is at most twice as large as the decrease in the expected probability of being alive. Since the expected probability of being alive can only decrease from 1 to 0, the expected gain (which starts at 0) can at most reach a value of 2 for the input string constructed in this way. Thus, for any randomized algorithm R that is restricted to choose at most one request per level, the expected value of the solution is $E[C(\mathcal{R})] \leq 2$.

If R is allowed to choose both requests from one level, it can reach at most an expected additional gain of 1 since it is out immediately after taking both requests from one level. This proves that, for any randomized algorithm R, the expected value of the solution computed by R is $E[C(\mathcal{R})] \leq 3$. □

We now investigate how many bits are needed to be communicated to an online algorithm A to yield optimality. Trivially, an upper bound is n, i. e., one bit is given for every request indicating whether it is part of an optimal solution or not.

As a next step, we look at a lower bound on the number of bits needed to achieve a particular competitive ratio.

Theorem 8 *For any online algorithm with advice, at least* $\frac{n+2}{2r} - 2$ *bits of advice are required to achieve a competitive ratio of* r.

Proof. Again, let $I_h \in \mathcal{I}_h$ be an input instance and let $\mathcal{T}$ denote the corresponding binary tree. Let A be an online algorithm that reads $b < h$ bits from the advice tape on any input I_h.

Obviously, b bits allow A to distinguish between 2^b different inputs, whereas there actually are 2^h inputs in $\mathcal{I}_h$. Applying the pigeonhole principle gives that some advice string ϕ is used for at least 2^{h-b} different inputs. For the ease of presentation, let use call the corresponding leaves in $\mathcal{T}$ *sinks*. Let $x_{i,k}$ (where $1 \leq k \leq 2^i$ for $0 \leq i \leq h$) denote the vertices of $\mathcal{T}$ on level i and let $\xi_{i,k}$ denote the number of sinks reachable from $x_{i,k}$. Since A is a deterministic algorithm, it makes a unique decision on every level $i+1$, depending only on the advice string ϕ and the input at hand, which is uniquely described by $x_{i,k}$. In the sequel, we show that there exists a sink whose corresponding input is hard for A.

Assume that we have already constructed some prefix of the input that corresponds to some vertex $x_{i,k}$. Consider the behavior of A for the constructed input prefix $x_{i,k}$ and advice ϕ. Depending on the behavior of A at level $i+1$, we distinguish the following cases:

1. A *satisfies one request at level* $i+1$*:* In this case, $C_{i+1}(\mathtt{A}(I_h)) = C_i(\mathtt{A}(I_h)) + 1$. Without loss of generality, let A satisfy the left request. Note that $x_{i+1,2k-1}$ is the left child of $x_{i,k}$ and $x_{i+1,2k}$ is the right child of $x_{i,k}$. We have two subcases:
   (a) If $\xi_{i+1,2k} = 0$, we are forced to extend the constructed input to $x_{i+1,2k-1}$, since we have already fixed the advice string ϕ.
   (b) If $\xi_{i+1,2k} \geq 1$, we extend the input to any sink in the subtree of $x_{i+1,2k}$, i. e., A satisfies the left request at level $i+1$, but we have just chosen the input such that the optimal algorithm satisfies the right request. Hence, A is out after processing the request at level $i+1$.
2. A *satisfies both requests at level* $i+1$*:* In this case, $C_{i+1}(\mathtt{A}(I_h)) = C_i(\mathtt{A}(I_h)) + 2$, but A is out after that, for it is not able to satisfy anymore requests henceforth, so we extend the input to any sink.
3. A *satisfies no request at level* $i+1$*:* In this case, $C_{i+1}(\mathtt{A}(I_h)) = C_i(\mathtt{A}(I_h))$, but on the other hand, A is able to satisfy any request on the next level. We extend the constructed input to $\xi_{i+1,k'} \geq \frac{\xi_{i,k}}{2}$, where $k' = 2k-1$ or $k' = 2k$, i. e., we choose the subtree of $x_{i,k}$ with a larger number of sinks.

Suppose that the case 1 (a) occurs c times and that $f \in \{0, \ldots, h\}$ is largest number such that A is not out after processing request at level f. Note that, while processing first f requests, only the cases 1 (a) and 3 occurred, because otherwise, A is already out after processing level f. Let $e = h - f$. It holds that $\xi_{0,1} = 2^{h-b}$, and case 3 occurred exactly $f - c$ times at the first f levels. Whenever case 3 occurred, the number of sinks in the subtree of

the processed input prefix decreased by at most one half. Whenever case 1 (a) occurred, the number of sinks did not decrease at all. Hence, after processing level f there are at least

$$2^{h-b} \cdot \left(\frac{1}{2}\right)^{h-e-c} \tag{12}$$

and at most 2^e sinks (which directly follows by the definition of e). This gives

$$b \geq c, \tag{13}$$

which formally proves the intuitive idea that at least one bit is needed for every decision that increases the number of chosen requests and also allows A not to be declared out.

Observe that an optimal online algorithm is able to satisfy exactly $n/2+1$ requests (one on every level, but both requests on level $h+1$). However, A makes exactly c right decisions, so, due to (8), it satisfies at most $c+2$ requests. We now want to guarantee a competitive ratio smaller than some r depending on b. By some easy calculations, we get that

$$r \geq \frac{\frac{n+2}{2}}{c+2}$$

and thus

$$c+2 \geq \frac{n+2}{2r}.$$

Equation (13) implies

$$b+2 \geq \frac{n+2}{2r},$$

which means that at least $\frac{n+2}{2r} - 2$ bits are necessary, thus proving the claimed bound. □

Note that, by setting $r = 1$, we immediately get a lower bound on the advice complexity for achieving an optimal solution.

Corollary 1. *For any online algorithm with advice, at least*

$$b_{opt} = \frac{n-2}{2}$$

advice bits are required to compute an optimal solution for DisPathAlloc.

Now, we present an online algorithm with advice for DisPathAlloc whose advice complexity is only a factor of $\log n$ away from this lower bound for large r, and is asymptotically tight for constant r.

Theorem 9 *For every r, there exists an r-competitive online algorithm for* DisPathAlloc *with advice complexity*

$$s(n) := \left(\frac{n}{r} + 3\right) \log n + \mathcal{O}(1),$$

or

$$s(n) := n \log \frac{r}{(r-1)^{\frac{r-1}{r}}} + 3 \log n + \mathcal{O}(1),$$

whichever is smaller.

Proof. Consider a very simple online algorithm A with advice that reads one bit of advice per request, and satisfies the request if and only if this bit is one. Since the optimal solution of any instance of length n is at most n, communicating an advice string with at most n/r (i. e., at least $n - n/r$ zeros) is sufficient to achieve an approximation ratio r. Hence, using Lemma 1 for $t := r$ implies the claim of the theorem. □

The previous theorem also shows that advice complexity $\mathcal{O}(\log^2 n)$ is sufficient to obtain a competitive ratio asymptotically better than any randomized algorithm is able to achieve. Indeed, for $r = n/\log n$, $\mathcal{O}(\log^2 n)$ advice bits are sufficient.

4.3 Advice Complexity Bounds with respect to the Size of the Line Network

Instead of considering the number of requests, we can also take the size of the input line network as a basis for measuring the advice complexity. In the following, we show that, also with respect to this measure, a linear number of advice bits is required for computing an optimal solution for DisPathAlloc.

Theorem 10 *Let* A *be a deterministic online algorithm with advice for* DisPathAlloc. *Then at least $L/2$ bits of advice are needed by* A *to achieve optimality.*

Proof. Let L be even. Consider the following set of inputs which basically consist of two phases. In the first $L/2$ time steps (phase 1), the paths $P_1 = (v_0, v_2), P_2 = (v_2, v_4), \ldots, P_{(L-1)/2} = (v_{L-2}, v_L)$ are requested. Phase 2 consists of $L/2$ rounds. Every round j may either be empty (no request is made) or consist of two consecutive requests P_{j_1} and P_{j_2} such that P_{j_1} and P_{j_2} are edge-disjoint subpaths of P_j. An example is shown in Fig. 8 for $L = 16$.

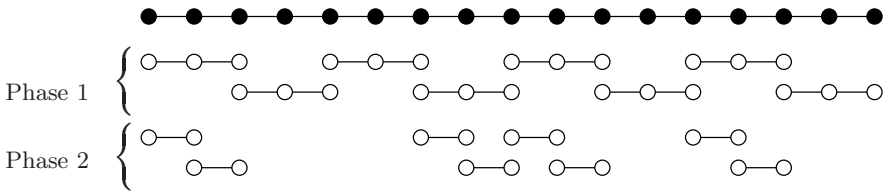


Fig. 8. Example input instance for DisPathAlloc

Obviously, for any j, the requests P_{j_1} and P_{j_2} can only be satisfied if P_j has been discarded before. Since phase 1 is always the same for all inputs, different inputs can be represented by a bit vector $B = (b_1, b_2, \ldots, b_{\frac{L}{2}})$ of length $L/2$ where b_j indicates whether P_{j_1} and P_{j_2} exist or not. Clearly, any online algorithm A has to behave differently for every string to achieve optimality in any case. The pigeonhole principle then directly implies that at least $L/2$ bits are needed to distinguish between all cases. □

Furthermore, a randomized online algorithm B with a competitive ratio of $\log L$ is given in [2]. We now investigate how many bits are needed to communicate all random decisions to a deterministic online algorithm A to be exactly as good as B in any case, i. e., to make exactly the same decisions. The idea of B is to cluster the line into $\log L$ different groups of possible requests and only accept calls which are within a certain group. Since the only time randomness is employed is when the group is chosen, it suffices to communicate a number between 1 and $\log L$ to be as good as B. Therefore, at most $\log \log L$ bits are needed (recall that the value of L is known to the algorithm).

5 Job Shop Scheduling

We will now consider the online version of the problem JSSchedule (short for *Job Shop Scheduling for two jobs*, see, e. g., [6, 7]). Consider n different machines $M_1, \ldots, M_n$. Every job A that has to be scheduled on these machines consist of n *tasks* $A_1, \ldots, A_n$, such that every task A_i has to be performed on a different machine. The tasks have to be completed in the given order, but the order of machines needed for this might differ from one job to another. Thus, we can represent a job A by a permutation $(i_1, \ldots, i_n)$ of the machine indices. Additionally, the following conditions have to hold:

1. All n tasks of A must be performed in the given order from A_1 to A_n.
2. Task A_j must be performed on machine M_{i_j}.
3. The execution of any task on its machine costs exactly one time unit.

In the following, we will restrict our attention to the case where there are exactly two jobs A, represented by $(i_1, \ldots, i_n)$, and B, represented by $(j_1, \ldots, j_n)$, which have to be scheduled. As in [6, 3], we use the following graphical view of the problem (see Fig. 9). Consider an $(n \times n)$-grid where the k-th column is labeled i_k and the l-th row is labeled j_l. We call the intersection of row l and column k a *cell* and, if $i_k = j_l$, this cell is called a *collision* because, in our model, this means that the same machine ought to process the i_k-th task of A and the j_l-th task of B. In our graphical presentation, we depict these collisions as black cells in the grid. At an arbitrary position in time, if the i_k-th task of A and the j_l-th task of B have to be processed, this can only happen in parallel if $i_k \neq j_l$. If $i_k = j_l$, however, both tasks need to be processed at the same time by the same machine. Starting at a point where no tasks have been processed yet (the top left corner in the grid), in every time step at least one task may be completed (and at most two). This means that, if a task of A is completed in a time step, we move one cell to the right in our grid. Analogously, if a task of B is completed in a time step, we move one cell down. However, if the two tasks for this time step do not require to run on the same machine, then we can just move down and to the right, meaning that both tasks are dealt with in this time step and do not need to be considered anymore in the following time steps. If all tasks are completed, we end up in the bottom right corner of the grid. We can therefore depict a correct schedule as a path in the grid from the top left corner to the bottom right corner where a cell may be traversed diagonally if and only if it contains no collision. However, if the path reaches the top left corner of a collision in the grid — we also say that it *hits an obstacle* in this case — it has to move around this obstacle by going to the right or going down which means that only one of the two tasks is processed at this time step.

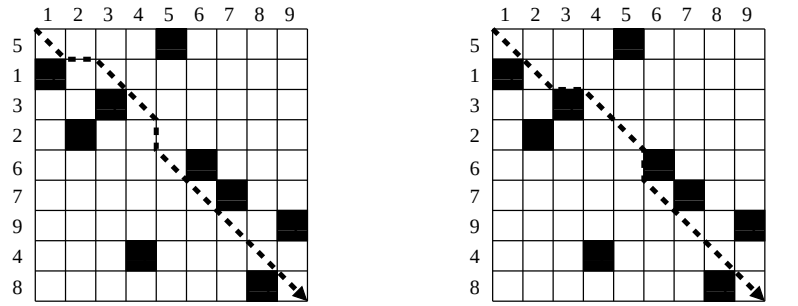
Clearly, a solution is considered optimal if it does not use more vertical and horizontal steps than any other solution. Let d denote the number of diagonal moves in the grid, h the number of moves to the right (i. e., horizontal moves), and v the number of moves down (i. e., vertical moves). Since we consider a quadratic grid and the path starts at the top left corner, eventually arriving at the bottom right corner, it directly follows that $h = v$ and that, overall, $d + h = d + v$ moves are needed.

In the following, we consider an online environment, i. e., a scenario where, in each time step i, the indices for the required machines for the two tasks A_i and B_i are revealed. At first, we want to investigate, what kind of information is necessary to achieve optimal output quality. We therefore prove the following lemma.

Lemma 4 *For every instance of* JSSCHEDULE*, there exists an optimal solution $\mathcal{A}$ which, as long as it does not arrive at the right or bottom border of the grid, always moves diagonally if no obstacle is in its way.*

Proof. We prove the lemma indirectly. Suppose that there is no such solution, which means that every optimal solution makes at least one move to the right or down without directly facing an obstacle in the next time step or having arrived at the border of the grid. Towards contradiction, let us impose a partial ordering $\prec$ on all optimal solutions where, for two solutions $\mathcal{B}$ and $\mathcal{C}$, it holds that $\mathcal{B} \prec \mathcal{C}$ iff $\mathcal{B}$ violates the above property for the first time in time step i, $\mathcal{C}$ violates the above property in time step j for the first time and $i < j$. Let $\mathcal{A}$ be any optimal solution that is maximal with respect to $\prec$ and let v denote the number of moves of $\mathcal{A}$ towards the bottom, and h the number of moves to the right. We represent $\mathcal{A}$'s moves by a vector $m_{\mathcal{A}} \in \{D, V, H\}^{n+v}$ where $(m_{\mathcal{A}})_i = D$ ($(m_{\mathcal{A}})_i = V$, $(m_{\mathcal{A}})_i = H$) means that, at time step i, $\mathcal{A}$ performs a diagonal (vertical, horizontal) move, respectively.

Let l be the first time step where $\mathcal{A}$ violates the above property, i. e. it does not make a diagonal move although it could. Without loss of generality, let $\mathcal{A}$ do a move to the right at time step l. This means that, eventually, after a (possibly empty) sequence of Hs starting at $l+1$, at the k-th position (with $l+1 \leq k$) in $m_{\mathcal{A}}$, there appears a V or a D (otherwise, since $\mathcal{A}$ is feasible, it would have already arrived at the bottom border of the grid). In case of a V, we may exchange the H at position l by a D and delete the V at position k, thus decreasing the costs of $\mathcal{A}$. Therefore, this appearance contradicts the optimality of $\mathcal{A}$. If, however, a D appears at position k, we may switch the moves at positions l and k which obviously does not increase the costs and thereby creates an optimal solution $\mathcal{A}'$ with $\mathcal{A} \prec \mathcal{A}'$, contradicting the maximality of $\mathcal{A}$ with respect to $\prec$. □



(a) An arbitrary optimal solution

(b) A solution as described by Lemma 4

Fig. 9. The graphical model of an instance of JSSCHEDULE and two optimal solutions

An example of a solution satisfying the property described in Lemma 4 is shown in Fig. 9 (b). Next, we need the following observation.

Lemma 5 *For every instance of* JSSCHEDULE*, there exists an optimal solution with costs at most $n + \lceil \sqrt{n} \rceil$.*

For a proof, we refer to [6]. Consider the optimal solution for an arbitrary instance, and assume that it makes d diagonal moves, h moves to the right, and v moves down. The cost

of this solution is $d + 2h = n + h$, since $d + h = n$. Hence, Lemma 5 implies that $h \leq \lceil \sqrt{n} \rceil$. Furthermore, using Lemma 4, it not difficult to see that there exists an online algorithm A with advice needing at most $2h \leq 2\lceil \sqrt{n} \rceil$ bits of advice: Algorithm A simply acts according to Lemma 4. Only at time steps where a diagonal move is not possible due to an obstacle, it has to read one bit from the advice tape indicating whether to move down or to the right.

Corollary 2 *For every instance of* JSSCHEDULE*, there exists an optimal online algorithm* A *with advice complexity* $s(n) = 2\lceil \sqrt{n} \rceil$.

Next, we will look at lower bounds on the number of bits necessary to create optimal output by an online algorithm.

Theorem 11 *Any online algorithm* A *with advice for* JSSCHEDULE *needs advice complexity* $s(n) = \Omega(\sqrt{n})$ *to achieve optimality.*

Proof. Let k be an arbitrary odd integer. Consider a *widget* that consists of four diagonal blocks placed orthogonally to the main diagonal. The first and third block have length $k+1$, the second and fourth block have length k. Exactly one of the middle obstacles (since $k+1$ is even, there are two of them) of the third block is missing, creating a *hole* in it. The widget then contains one more row and column with one obstacle each to make the widget a permutation matrix. Hence, the widget has size $(4k+3) \times (4k+3)$. An example of the widget for $k = 5$ is depicted in Fig. 10.

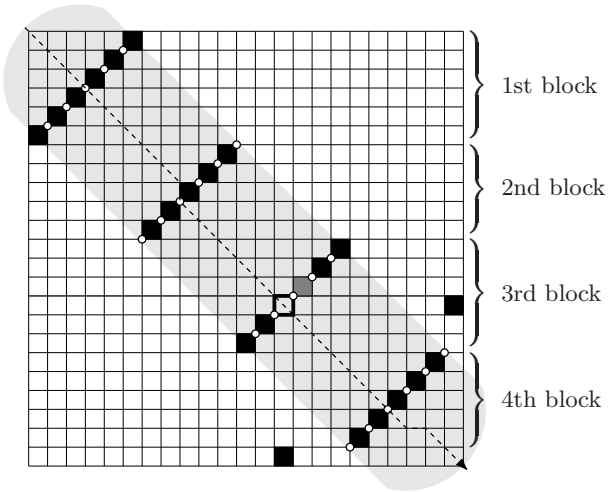


Fig. 10. The employed widget. The thick-framed empty cell is the hole, the gray square is the alternative position of the hole. The small circles are points in the segments. The light-gray area represents the active zone. The dashed line shows the part of the optimal solution passing through the widget.

The set of points between adjacent obstacles of the i-th block of the widget is called the i-th *segment* of the widget. (In case of blocks of size k, we consider the corners of the block to be parts of the i-th segment, too.)

Consider an input instance I constructed by concatenating k widgets. In this case, the size of the input instance I is $n = (4k+3)k$. We call the area of the input instance parallel to the main diagonal, that encloses all segments of all widgets, the *active zone*.

It is not difficult to see that there is a solution of I with cost $n+k$ which follows the main diagonal until it hits an obstacle of the second block of some widget. There it moves either to the right or down, depending on the location of the hole in third block. Afterwards, it moves diagonally through the hole, without hitting any obstacle. At the end of the widget, it makes a move down or right to return to the main diagonal. In this way, there are exactly k moves down and k moves to the right, so the cost of this solution is exactly $n+k$.

Next, we show that no optimal solution leaves the active zone. Indeed, to leave it, more than k moves down or more than k moves to the right are necessary. But any such solution has cost of at least $n+k+1$ and is therefore not optimal.

Now consider any solution $\mathcal{A}$ of I that does not leave the active zone, and consider the i-th widget of the input. We claim that $\mathcal{A}$ makes at least one non-diagonal move between the first and second segment of this i-th widget and, unless $i=k$, at least one non-diagonal move between the fourth segment of the i-th widget and the first segment of the $(i+1)$-th widget. Indeed, since $\mathcal{A}$ does not leave the active zone, it must pass through some point of the first segment of the i-th widget, which has an even distance (using *Manhattan metric*) to the main diagonal. Eventually, it has to pass through some point of the second segment, which has an odd distance to the main diagonal. Since the distance to the main diagonal changes only with non-diagonal moves, there must be at least one such move. The same argument holds for the second part of the claim.

We have shown that any solution of I makes at least $2k-1$ non-diagonal moves, hence its cost is at least $n+k$. Furthermore, any optimal solution does not make any non-diagonal moves between the second and the fourth segment of any widget.

Now we are ready to show that any online algorithm `A` that solves each k-widget input instance optimally needs at least k bits of advice. By contradiction, assume that less than k bits of advice are sufficient for some `A`. Since there are 2^k different k-widget inputs (because there are two different possibilities to place the hole in each widget), there are two different inputs I_1, I_2 with the same advice. Assume that the first widget where I_1 and I_2 differ is the i-th one. Since `A` uses the same advice in I_1 and I_2, it reaches the same point on the second segment of the i-th widget in both I_1 and I_2. However, the only possibility to not make any non-diagonal move until the fourth segment of the i-th widget is to pass through the hole. As the hole is located differently in I_1 and I_2, this cannot happen in both I_1 and I_2. Hence, `A` behaves non-optimally in at least one of these instances.

To conclude the proof, note that for any n, we can pick the largest odd k such that $(4k+3)k \leq n$, i. e.,

$$k \geq \left\lfloor \frac{-3+\sqrt{9+16n}}{8} \right\rfloor - 1.$$

It is straightforward to check that any algorithm `A` that solves each input of size n optimally needs at least as many bits of advice as any algorithm solving each input of size $(4k+3)k \leq n$ optimally. Indeed, any input of smaller size can be padded to a larger size by a set of obstacles that keeps the main diagonal free in the padded region (so the padding does not increase the cost of the optimal solution). Therefore, the advice complexity of any optimal online algorithm for JSSCHEDULE is $s(n) = \Omega(\sqrt{n})$. □

We conclude this section by comparing randomized algorithms and algorithms with advice. In [6], a randomized algorithm for JSSCHEDULE with a competitive ratio of $\mathcal{O}(1+1/\sqrt{n})$ is presented. The idea of the algorithm is to pick one diagonal uniformly at random and follow it. Whenever an obstacle is reached, the algorithm goes around it by making one move to the

right and one move down. Since there are only $2n-1$ different diagonals, the randomized algorithm uses at most $1+\log(n)$ random bits. Easily, this implies the following fact.

Fact 5 *There is an online algorithm for* JSSCHEDULE *with advice complexity* $s(n) \leq 1+\log(n)$ *that achieves competitive ratio* $\mathcal{O}(1+1/\sqrt{n})$.

6 Conclusion

We proposed a refined model of advice complexity for online problems and analyzed three well-known online problems with respect to this complexity measure. We demonstrated that in these cases very small advice (logarithmic number of bits, for paging even a constant number of bits) suffices to significantly decrease the competitive ratio in comparison to any deterministic online algorithm. On the other hand, some hard online problems like paging and disjoint path allocation require a linear number of advice bits for computing an optimal solution.

Furthermore, the comparison of online algorithms with advice to randomized online algorithms shows that for some problems a logarithmic number of advice bits is sufficient to achieve the same competitive ratio than any randomized algorithm, as it is the case for all problems analyzed in this paper. Hence, one might suggest a hypothesis that small advice complexity is sufficient to achieve the competitive ratio of the best randomized algorithm for *any* online problem. This claim, however, cannot be proven in full generality. In fact, we can construct a online problem where the number of required advice bits is as high as the number of random bits in order to keep up with randomized algorithms.

Consider the following problem, where the online algorithm has to guess a sequence of n bits. If all bits are guessed correctly, the algorithm is greatly rewarded, otherwise, the cost of the solution is zero. More formally, the input is a sequence $(x_1, x_2, \ldots, x_{n+1})$, where x_i is either 0 or 1 for each $i \leq n$ and x_{n+1} is either $\overline{0}$ or $\overline{1}$. For each request x_i, $i \leq n$, the online algorithm has to specify one bit y_i, its guess about the next bit x_{i+1}. The cost of the solution is 2^n if $y_i = x_{i+1}$ or $y_i = \overline{x_{i+1}}$ for all $i \leq n$, and 0 otherwise.[4] Let R be a randomized algorithm that just tosses a coin for each decision. Easily, the expected cost of the solution computed by R is one, and algorithm R uses n random bits (since the last request is marked, R does not need to toss a coin there). For sure, n advice bits are sufficient to solve the problem optimally. If, however, we only allow $n-1$ advice bits, then an adversary can always choose an input sequence that is not chosen by A, resulting in the cost zero.

Hence, we have exhibited an online problem for which a linear number of advice bits is needed to outperform a randomized algorithm. Unfortunately, this situation is quite artificial: If an algorithm with advice A outperforms a randomized algorithm R, then A is exponentially better than R, too. Nevertheless, this situation shows that, in general, the advice complexity is somehow orthogonal to randomization. It remains as an open problem to find more connections between these complexity measures, e.g., to somehow characterize classes of online problems where small advice is sufficient to keep up with randomized algorithms. Furthermore, it might be interesting to consider randomized online algorithms with advice.

[4] One could also define an iterated version of this problem, where input instances consist of several delimited blocks, and the algorithm scores $2^{|b_i|}$ for each block b_i where all numbers were guessed correctly. In this way, the algorithm can have some information about the cost of the solution during the computation. Although this alternative definition may be slightly more natural, it does not change the core of the problem.

Acknowledgments

The authors would like to thank Juraj Hromkovič for many helpful discussions on the topic of this paper and elaborating the model of *Online Algorithms with Advice* to the point we could start investigating the problems introduced.

References

1. S. Ben-David and A. Borodin. A new measure for the study of on-line algorithms. *Algorithmica*, 11(1):73–91, 1994.
2. A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, 1998.
3. P. Brucker. An efficient algorithm for the job-shop problem with two jobs. *Computing*, 40(4):353–359, 1988.
4. S. Dobrev, R. Královič, and D. Pardubská. How much information about the future is needed? In *34th Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM'08)*, pp. 247–258, 2008.
5. J. Hromkovič. *Theoretical Computer Science: Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography.* Springer-Verlag, 2004.
6. J. Hromkovič. *Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms.* Springer-Verlag New York, 2006.
7. J. Hromkovič, T. Mömke, K. Steinhöfel, P. Widmayer. Job shop scheduling with unit length tasks: bounds and algorithms. *Algorithmic Operations Research*, 2(1):1–14, 2007.
8. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.