BRICS LS-96-2 S. Albers: Competitive Online Algorithms

BRICS

Basic Research in Computer Science

# Competitive Online Algorithms

Susanne Albers

BRICS Lecture Series LS-96-2

ISSN 1395-2048 September 1996

**See back inner page for a list of recent publications in the BRICS Lecture Series. Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through WWW and anonymous FTP:**

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

# *Competitive Online Algorithms*

*Susanne Albers*

Susanne Albers
Max-Planck-Institut
Saarbrücken
Germany

# Preface

I would like to thank BRICS and, in particular, Erik Meineche Schmidt for giving me the opportunity to teach a mini-course on competitive online algorithms at Aarhus University, August 27–29, 1996. Lecturing the course was a great pleasure for me. In winter 1995/96, Dany Breslauer first pointed out to me the possibility to teach a mini-course in Århus, and I thank him for his encouragement. Furthermore, I would like to thank all members of the Computer Science Department at Aarhus University for their very kind hospitality throughout my stay.

The mini-course consisted of three lectures. In the first lecture we gave basic definitions and presented important techniques that are used in the study on online algorithms. The paging problem was always the running example to explain and illustrate the material. We also discussed the $k$-server problem, which is a very well-studied generalization of the paging problem.

The second lecture was concerned with self-organizing data structures, in particular self-organizing linear lists. We presented results on deterministic and randomized online algorithms. Furthermore, we showed that linear lists can be used to build very effective data compression schemes and reported on theoretical as well as experimental results.

In the third lecture we discussed three application areas in which interesting online problems arise. The areas were (1) distributed data management, (2) scheduling and load balancing, and (3) robot navigation and exploration. In each of these fields we gave some important results.

# Contents

# 1 Online algorithms and competitive analysis

## 1.1 Basic definitions

Formally, many online problems can be described as follows. An online algorithm $A$ is presented with a *request sequence* $\sigma = \sigma(1)$, $\sigma(2), \ldots, \sigma(m)$. The algorithm $A$ has to serve each request *online*, i.e., without knowledge of future requests. More precisely, when serving request $\sigma(t)$, $1 \le t \le m$, the algorithm does not know any request $\sigma(t')$ with $t' > t$. Serving requests incurs cost, and the goal is to serve the entire request sequence so that the total cost is as small as possible. This setting can also be regarded as a *request-answer* game: An adversary generates requests, and an online algorithm has to serve them one at a time.

In order to illustrate this formal model, we mention a concrete online problem.

**The paging problem**: Consider a two-level memory system that consists of a small fast memory and a large slow memory. Here, each request specifies a page in the memory system. A request is served if the corresponding page is in fast memory. If a requested page is not in fast memory, a *page fault* occurs. Then a page must be moved from fast memory to slow memory so that the requested page can be loaded into the vacated location. A paging algorithms specifies which page to evict on a fault. If the algorithm is online, then the decision which page to evict must be made without knowledge of any future requests. The cost to be minimized is the total number of page faults incurred on the request sequence.

Sleator and Tarjan [48] suggested to evaluate the performance on an online algorithm using *competitive analysis*. In a competitive analysis, an online algorithm $A$ is compared to an *optimal offline algorithm*. An optimal offline algorithm knows the entire request sequence in advance and can serve it with minimum cost. Given a request sequence $\sigma$, let $C_A(\sigma)$ denote the cost incurred by $A$ and let $C_{OPT}(\sigma)$ denote the cost paid by an optimal offline algorithm OPT. The algorithm $A$ is called $c$-competitive if there exists a constant $a$ such that

$$C_A(\sigma) \leq c \cdot C_{OPT}(\sigma) + a$$

for all request sequences $\sigma$. Here we assume that $A$ is a deterministic online algorithm. The factor $c$ is also called the *competitive ratio* of $A$.

## 1.2 Results on deterministic paging algorithms

We list three well-known deterministic online paging algorithms.

- **LRU** (Least Recently Used): On a fault, evict the page in fast memory that was requested least recently.
- **FIFO** (First-In First-Out): Evict the page that has been in fast memory longest.
- **LFU** (Least Frequently Used): Evict the page that has been requested least frequently.

Before analyzing these algorithms, we remark that Belady [12] exhibited an optimal offline algorithm for the paging problem. The algorithm is called MIN and works as follows.

- **MIN**: On a fault, evict the page whose next request occurs furthest in the future.

Belady showed that on any sequence of requests, MIN achieves the minimum number of page faults.

Throughout these notes, when analyzing paging algorithms, we denote by $k$ the number of pages that can simultaneously reside in fast memory. It is not hard to see that the algorithm LFU is not competitive. Sleator and Tarjan [48] analyzed the algorithms LRU and FIFO and proved the following theorem.

**Theorem 1** *The algorithms LRU and FIFO are $k$-competitive.*

**Proof:** We will show that LRU is $k$-competitive. The analysis for FIFO is very similar. Consider an arbitrary request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$. We will prove that $C_{LRU}(\sigma) \leq k \cdot C_{OPT}(\sigma)$. Without loss of generality we assume that LRU and OPT initially start with the same fast memory.

We partition $\sigma$ into phases $P(0), P(1), P(2), \ldots$ such that LRU has at most $k$ fault on $P(0)$ and exactly $k$ faults on $P(i)$, for every $i \geq 1$. Such a partitioning can be obtained easily. We start at the end of $\sigma$ and scan the request sequence. Whenever we have seen $k$ faults made by LRU, we cut off a new phase. In the remainder of this proof we will show that OPT has at least one page fault during each phase. This establishes the desired bound.

For phase $P(0)$ there is nothing to show. Since LRU and OPT start with the same fast memory, OPT has a page fault on the first request on which LRU has a fault.

Consider an arbitrary phase $P(i)$, $i \geq 1$. Let $\sigma(t_i)$ be the first request in $P(i)$ and let $\sigma(t_{i+1} - 1)$ be the last request in $P(i)$. Furthermore, let $p$ be the page that is requested last in $P(i-1)$.

**Lemma 1** *$P(i)$ contains requests to $k$ distinct pages that are different from $p$.*

If the lemma holds, then OPT must have a page fault in $P(i)$. OPT has page $p$ in its fast memory at the end of $P(i-1)$ and thus cannot have all the other $k$ pages request in $P(i)$ in its fast memory.

It remains to prove the lemma. The lemma clearly holds if the $k$ requests on which LRU has a fault are to $k$ distinct pages and if these pages are also different from $p$. So suppose that LRU faults twice on a page $q$ in $P(i)$. Assume that LRU has a fault on $\sigma(s_1) = q$ and $\sigma(s_2) = q$, with $t_i \le s_1 < s_2 \le t_{i+1} - 1$. Page $q$ is in LRU's fast memory immediately after $\sigma(s_1)$ is served and is evicted at some time $t$ with $s_1 < t < s_2$. When $q$ is evicted, it is the least recently requested page in fast memory. Thus the subsequence $\sigma(s_1), \ldots, \sigma(t)$ contains requests to $k+1$ distinct pages, at least $k$ of which must be different from $p$.

Finally suppose that within $P(i)$, LRU does not fault twice on page but on one of the faults, page $p$ is request. Let $t \ge t_i$ be the first time when $p$ is evicted. Using the same arguments as above, we obtain that the subsequence $\sigma(t_i - 1), \sigma(t_i), \ldots, \sigma(t)$ must contain $k+1$ distinct pages. □

The next theorem is also due to Sleator and Tarjan [48]. It implies that LRU and FIFO achieve the best possible competitive ratio.

**Theorem 2** *Let $A$ be a deterministic online paging algorithm. If $A$ is c-competitive, then $c \ge k$.*

**Proof:** Let $S = \{p_1, p_2, \ldots, p_{k+1}\}$ be a set of $k+1$ arbitrary pages. We assume without loss of generality that $A$ and OPT initially have $p_1, \ldots, p_k$ in their fast memories.

Consider the following request sequence. Each request is made to the page that is not in $A$'s fast memory.

Online algorithm $A$ has a page fault on every request. Suppose that OPT has a fault on some request $\sigma(t)$. When serving $\sigma(t)$, OPT can evict a page is not requested during the next $k-1$ requests $\sigma(t+1), \ldots, \sigma(t+k-1)$. Thus, on any $k$ consecutive requests, OPT has at most one fault. □

The competitive ratios shown for deterministic paging algorithms are not very meaningful from a practical point of view. Note that the performance ratios of LRU and FIFO become worse as the size of the fast memory increases. However, in practice, these algorithms perform better the bigger the fast memory is. Furthermore, the competitive ratios of LRU and FIFO are the same, whereas in practice LRU performs much better. For these reasons, there has been a study of competitive paging algorithms with *access graphs* [23, 36]. In an access graph, each node represents a page in the memory system. Whenever a page $p$ is requested, the next request can only be to a page that is adjacent to $p$ in the access graph. Access graphs can model more realistic request sequences that exhibit locality of reference. It was shown [23, 36] that using access graphs, one can overcome some negative aspects of conventional competitive paging results.

# 2 Randomization in online algorithms

## 2.1 General concepts

The competitive ratio of a randomized online algorithm $A$ is defined with respect to an adversary. The adversary generates a request sequence $\sigma$ and it also has to serve $\sigma$. When constructing $\sigma$, the adversary always knows the description of $A$. The crucial question is: When generating requests, is the adversary allowed to see the outcome of the random choices made by $A$ on previous requests?

Ben-David *et al.* [17] introduced three kinds of adversaries.

- **Oblivious Adversary**: The oblivious adversary has to generate a complete request sequence in advance, before any requests are served by the online algorithm. The adversary is charged the cost of the optimum offline algorithm for that sequence.
- **Adaptive Online Adversary**: This adversary may observe the online algorithm and generate the next request based on

the algorithm's (randomized) answers to all previous requests. The adversary must serve each request online, i.e., without knowing the random choices made by the online algorithm on the present or any future request.

- **Adaptive Offline Adversary**: This adversary also generates a request sequence adaptively. However, it is charged the optimum offline cost for that sequence.

A randomized online algorithm $A$ is called $c$-competitive against any oblivious adversary if there is a constant $a$ such for all request sequences $\sigma$ generated by an oblivious adversary, $E[C_A(\sigma)] \leq c \cdot C_{OPT}(\sigma) + a$. The expectation is taken over the random choices made by $A$.

Given a randomized online algorithm $A$ and an adaptive online (adaptive offline) adversary ADV, let $E[C_A]$ and $E[C_{ADV}]$ denote the expected costs incurred by $A$ and ADV in serving a request sequence generated by ADV. A randomized online algorithm $A$ is called $c$-competitive against any adaptive online (adaptive offline) adversary if there is a constant $a$ such that for all adaptive online (adaptive offline) adversaries ADV, $E[C_A] \leq c \cdot E[C_{ADV}] + a$, where the expectation is taken over the random choices made by $A$.

Ben-David *et al.* [17] investigated the relative strength of the adversaries with respect to an arbitrary online problem and showed the following statements.

**Theorem 3** *If there is a randomized online algorithm that is c-competitive against any adaptive offline adversary, then there also exists a c-competitive deterministic online algorithm.*

This theorem implies that randomization does not help against the adaptive offline adversary.

**Theorem 4** *If $A$ is a $c$-competitive randomized algorithm against any adaptive online adversary, and if there is a $d$-competitive algorithm against any oblivious adversary, then $A$ is $(c \cdot d)$-competitive against any adaptive offline adversary.*

An immediate consequence of the above two theorems in the following corollary.

**Corollary 1** *If there exists a $c$-competitive randomized algorithm against any adaptive online adversary, then there is a $c^2$-competitive deterministic algorithm.*

## 2.2 Randomized paging algorithms against oblivious adversaries

We will prove that, against oblivious adversaries, randomized online paging algorithms can considerably beat the ratio of $k$ shown for deterministic paging. The following algorithm was proposed by Fiat *et al.* [27].

**Algorithm MARKING**: The algorithm processes a request sequence in phases. At the beginning of each phase, all pages in the memory system are unmarked. Whenever a page is requested, it is *marked*. On a fault, a page is chosen uniformly at random from among the unmarked pages in fast memory, and this pages is evicted. A phase ends when all pages in fast memory are marked and a page fault occurs. Then, all marks are erased and a new phase is started.

Fiat *et al.* [27] analyzed the performance of the MARKING algorithm.

**Theorem 5** *The MARKING algorithm is $2H_k$-competitive against any oblivious adversary, where $H_k = \sum_{i=1}^{k} 1/i$ is the $k$-th Harmonic number.*

Note that $H_k$ is roughly $\ln k$. Later, in Section 3.2, we will see that no randomized online paging algorithm against any oblivious adversary can be better than $H_k$-competitive. Thus the MARKING algorithm is optimal, up to a constant factor. More complicated paging algorithms achieving an optimal competitive ratio of $H_k$ were given in [42, 1].

**Proof:** Given a request sequence $\sigma = \sigma(1), \ldots, \sigma(m)$, we assume without of generality that MARKING already has a fault on the first request $\sigma(1)$.

MARKING divides the request sequence into phases. A phase starting with $\sigma(i)$ ends with $\sigma(j)$, where $j$, $j > i$, is the smallest integer such that the set

$$\{\sigma(i), \sigma(i+1), \ldots, \sigma(j+1)\}$$

contains $k+1$ distinct pages. Note that at the end of a phase all pages in fast memory are marked.

Consider an arbitrary phase. Call a page *stale* if it is unmarked but was marked in the previous phase. Call a page *clean* if it is neither stale nor marked.

Let $c$ be the number of clean pages requested in the phase. We will show that

1. the amortized number of faults made by OPT during the phase it at least $\frac{c}{2}$.
2. the expected number of faults made by MARKING is at most $cH_k$.

These two statements imply the theorem.

We first analyze OPT's cost. Let $S_{OPT}$ be the set of pages contained in OPT's fast memory, and let $S_M$ be the set of pages stored in MARKING's fast memory. Furthermore, let $d_I$ be the value of

$|S_{OPT} \setminus S_M|$ at the beginning of the phase and let $d_F$ be the value of $|S_{OPT} \setminus S_M|$ at the end of the phase. OPT has at least $c - d_I$ faults during the phase because at least $c - d_I$ of the $c$ clean pages are not in OPT's fast memory. Also, OPT has at least $d_F$ faults during the phase because $d_F$ pages requested during the phase are not in OPT's fast memory at the end of the phase. We conclude that OPT incurs at least

$$\max\{c - d_I, d_F\} \geq \frac{1}{2}(c - d_I + d_F) = \frac{c}{2} - \frac{d_I}{2} + \frac{d_F}{2}$$

faults during the phase. Summing over all phases, the terms $\frac{d_I}{2}$ and $\frac{d_F}{2}$ telescope, except for the first and last terms. Thus the amortized number of page faults made by OPT during the phase is at least $\frac{c}{2}$.

Next we analyze MARKING's expected cost. Serving $c$ requests to clean pages cost $c$. There are $s = k - c \leq k - 1$ requests to stale pages. For $i = 1, \ldots, s$, we compute the expected cost of the $i$-th request to a stale page. Let $c(i)$ be the number of clean pages that were requested in the phase immediately before the $i$-th request to a stale page and let $s(i)$ denote the number of stale pages that remain before the $i$-th request to a stale page.

When MARKING serves the $i$-th request to a stale page, exactly $s(i) - c(i)$ of the $s(i)$ stale pages are in fast memory, each of them with equal probability. Thus the expected cost of the request is

$$\frac{s(i) - c(i)}{s(i)} \cdot 0 + \frac{c(i)}{s(i)} \cdot 1 \leq \frac{c}{s(i)} = \frac{c}{k - i + 1}.$$

The last equation follows because $s(i) = k - (i - 1)$. The total expected cost for serving requests to stale pages is

$$\sum_{i=1}^{s} \frac{c}{k + 1 - i} \leq \sum_{i=2}^{k} \frac{c}{i} = c(H_k - 1).$$

We conclude that MARKING's total expected cost in the phase is bounded by $cH_k$. □

# 3 Proof techniques

## 3.1 Potential functions

We present an important proof technique that can be used to develop upper bounds for deterministic and randomized online algorithms. Consider an online algorithm $A$. In a competitive analysis we typically want to show that for all request sequences $\sigma = \sigma(1), \ldots, \sigma(m)$,

$$C_A(\sigma) \leq c \cdot C_{OPT}(\sigma), \tag{1}$$

for some constant $c$. Assume for the moment that we deal with a deterministic online algorithm. Usually, a bound given in (1) cannot be established by comparing online and offline cost for each request separately. For instance, if we consider the paging problem and concentrate on a single request, it is possible that the online cost is 1 (the online algorithm has a page fault), whereas the optimal offline cost is 0 (the offline algorithm does not have page fault). Thus, on a single request, the ratio online cost/offline cost can be infinity. However, on the entire request sequence $\sigma$, the bound given in (1) might hold. This implies that *on the average*,

$$C_A(t) \leq c \cdot C_{OPT}(t)$$

holds for every request $\sigma(t)$, $1 \leq t \leq m$. Here $C_A(t)$ and $C_{OPT}(t)$ denote the actual costs incurred by $A$ and OPT on request $\sigma(t)$. In a competitive analysis, an averaging of cost among requests can be done using a *potential function*. We refer the reader to [50] for a comprehensive introduction to amortized analyses using potential functions.

Given a request sequence $\sigma = \sigma(1), \ldots, \sigma(m)$ and a potential function $\Phi$, the *amortized online cost* on request $\sigma(t)$, $1 \leq t \leq m$, is defined as $C_A(t) + \Phi(t) - \Phi(t-1)$. Here $\Phi(t)$ is the value of the potential function after request $\sigma(t)$, i.e., $\Phi(t) - \Phi(t-1)$ is the change in potential that occurs during the processing of $\sigma(t)$. In an amortized analysis using a potential function we usually show that for any request $\sigma(t)$,

$$C_A(t) + \Phi(t) - \Phi(t-1) \leq c \cdot C_{OPT}(t). \tag{2}$$

If we can prove this inequality for all $t$, then it is easy to see that $A$ is $c$-competitive. Summing up (2) for all $t = 1, \ldots, m$, we obtain

$$\sum_{t=1}^{m} C_A(t) + \Phi(m) - \Phi(0) \leq c \sum_{t=1}^{m} C_{OPT}(t), \tag{3}$$

where $\Phi(0)$ is the initial potential. Typically a potential function is chosen such that $\Phi$ is always non-negative and such that the initial potential is 0. Using these two properties, we obtain from inequality (3), as desired, $C_A(\sigma) \leq c \cdot C_{OPT}(\sigma)$.

The difficult part in a competitive analysis using a potential function is to construct $\Phi$ and show inequality (2) for all requests. If $A$ is a randomized online algorithm, then the expected amortized cost incurred by $A$ has to be compared to the cost of the respective adversary, for all requests $\sigma(t)$.

We now apply potential functions to give an alternative proof that LRU is $k$-competitive.

As usual let $\sigma = \sigma(1), \ldots, \sigma(m)$ be an arbitrary request sequence. At any time let $S_{LRU}$ be the set of pages contains in LRU's fast memory, and let $S_{OPT}$ be the set of pages contained in OPT's fast memory. Set $S = S_{LRU} \setminus S_{OPT}$. Assign integer weights from the range $[1, k]$ to the pages in $S_{LRU}$ such that, for any two pages $p, q \in S_{LRU}$, $w(p) < w(q)$ iff the last request to $p$ occurred earlier than the last request to $q$. Let

$$\Phi = \sum_{p \in S} w(p).$$

Consider an arbitrary request $\sigma(t) = p$ and assume without loss of generality that OPT serves the request first and that LRU serves second. If OPT does not have a page fault on $\sigma(t)$, then its cost is 0 and the potential does not change. On the other hand, if OPT has a page fault, then its cost is 1. OPT might evict a page that is in LRU's fast memory, in which case the potential increases. However, the potential can increase by at most $k$.

Next suppose that LRU does not have fault on $\sigma(t)$. Then its cost is 0 and the potential cannot change. If LRU has a page fault, its cost on the request is 1. We show that the potential decreases by at least 1. Immediately before LRU serves $\sigma(t)$, page $p$ is only in OPT's fast memory. By symmetry, there must be a page that is only in LRU's fast memory, i.e, the must exit a page $q \in S$. If $q$ is evicted by LRU during the operation, then the potential decreases by $w(q) \geq 1$. Otherwise, since $p$ is loaded into fast memory, $q$'s weight must decrease by 1, and thus the potential must decrease by 1.

In summary we have shown: Every time OPT has a fault, the potential increases by at most $k$. Every time LRU has a fault, the potential decreases by at least 1. We conclude that

$$C_{LRU}(t) + \Phi(t) - \Phi(t-1) \leq k \cdot C_{OPT}(t)$$

must hold.

## 3.2 Yao's minimax principle

In this section we describe a technique for proving lower bounds on the competitive ratio that is achieved by randomized online algorithms against oblivious adversaries. The techniques is an application of Yao's minimax theorem [53]. In the context of online algorithms, Yao's minimax theorem can be formulated as follows: Given an online problem, the competitive ratio of the best randomized online algorithm against any oblivious adversary is equal to the competitive ratio of the best deterministic online algorithm under a worst-case input distribution.

More formally, let $c_R$ denote the smallest competitive ratio that is achieved by randomized online algorithm $R$ against any oblivious adversary. Furthermore, let $P$ be a probability distribution for choosing a request sequence. Given a deterministic online algorithm $A$, we denote by $c_A^P$ the smallest competitive ratio of $A$ under $P$, i.e., $c_A^P$ is the infimum of all $c$ such that $E[C_A] \leq c \cdot E[C_{OPT}] + a$. Here $E[C_A]$ and $E[C_{OPT}]$ denote the expected costs incurred by $A$ and OPT on request sequences that are generated according to $P$. As usual, the constant $a$ may not depend on the request sequence $\sigma$. Yao's minimax principle implies that

$$\inf_R c_R = \sup_P \inf_A c_A^P. \tag{4}$$

On the left-hand side of the equation, the infimum is taken over all randomized online algorithms. On the right-hand side, the supremum is taken over all probability distributions for choosing a request sequence and the infimum is taken over all deterministic online algorithms.

Using equation (4), we can construct a lower bound on the competitiveness of randomized online algorithms as follows: First we explicitly construct a probability distribution $P$. Then we develop a lower bound on $\inf_A c_A^P$ for any deterministic online algorithm $A$.

We can apply this strategy to prove a lower bound for randomized online paging algorithms.

**Theorem 6** *If $R$ is a randomized online paging algorithm that is $c$-competitive against any oblivious adversary, then $c \geq H_k$.*

The theorem was first proved by Fiat *et al.* [27]. Here we give an alternative proof presented in [43].

**Proof:** Let $S = \{p_1, \ldots, p_{k+1}\}$ be a set of $k+1$ pages. We construct a probability distribution for choosing a request sequence. The first request $\sigma(1)$ is made to a page that is chosen uniformly at random from $S$. Every other request $\sigma(t)$, $t > 1$, it made to a page that is chosen uniformly at random from $S \setminus \{\sigma(t-1)\}$.

We partition the request sequence into phases. A phase starting with $\sigma(i)$ ends with $\sigma(j)$, where $j$, $j > i$, is the smallest integer such that

$$\{\sigma(i), \sigma(i+1), \ldots, \sigma(j+1)\}$$

contains $k+1$ distinct pages.

OPT incurs one page fault during each phase.

Consider any deterministic online paging algorithm $A$. What is the expected cost of $A$ on a phase? The expected cost on each request is $\frac{1}{k}$ because at any time exactly one page is not in $A$'s fast memory. The probability that the request sequence hits this page is $\frac{1}{k}$. We have to determine the expected length of the phase and study a related problem. Consider a random walk on the complete graph $K_{k+1}$. We start at some vertex of the graph and in each step move to a neighboring vertex; each neighboring vertex is chosen with equal probability. Clearly, the expected number steps until all vertices are visited is equal to the expected length of the phase. A well-known result in the theory of random walks states that the expected number of steps to visit all vertices is $kH_k$. □

# 4 The $k$-server problem

In the $k$-server problem we are given a metric space $S$ and $k$ mobile servers that reside on points in $S$. Each request specifies a point $x \in S$. To serve a request, one of the $k$ server must be moved to the requested point unless a server is already present. Moving a server from point $x$ to point $y$ incurs a cost equal to the distance between

$x$ and $y$. The goal is to serve a sequence of requests so that the total distance traveled by all serves is as small as possible.

The $k$-server problem contains paging as a special case. Consider a metric space in which the distance between any two points in 1; each point in the metric space represents a page in the memory system and the pages covered by servers are those that reside in fast memory. The $k$-server problem also models more general caching problem, where the cost of loading an item into fast memory depends on the size of the item. Such a situation occurs, for instance, when font files are loaded into the cache of printers. More generally, the $k$-server problem can also be regarded as a vehicle routing problem.

The $k$-server problem was introduced by Manasse *et al.* [41] in 1988 who also showed a lower bound for deterministic $k$-server algorithms.

**Theorem 7** *Let $A$ be a deterministic online $k$-server algorithm in a arbitrary metric space. If $A$ is $c$-competitive, then $c \geq k$.*

**Proof:** We will construct a request sequence $\sigma$ and $k$ algorithms $B_1, \ldots, B_k$ such that

$$C_A(\sigma) = \sum_{j=1}^{k} C_{B_j}(\sigma).$$

Thus, there must exist a $j_0$ such that $\frac{1}{k} C_A(\sigma) \geq C_{B_{j_0}}(\sigma)$. Let $S$ be the set of points initially covered by $A$'s servers plus one other point. We can assume that $A$ initially covers $k$ distinct points so that $S$ has cardinality $k+1$.

A request sequence $\sigma = \sigma(1), \ldots, \sigma(m)$ is constructed in the following way. At any time a request is made to the point not covered by $A$'s servers.

For $t = 1, \ldots, m$, let $\sigma(t) = x_t$. Let $x_{m+1}$ be the point that is finally uncovered. Then

$$C_A(\sigma) = \sum_{t=1}^{m} dist(x_{t+1}, x_t) = \sum_{t=1}^{m} dist(x_t, x_{t+1}).$$

Let $y_1, \ldots, y_k$ be the points initially covered by $A$. Algorithm $B_j$, $1 \leq j \leq k$, is defined as follows. Initially, $B_j$ covers all points in $S$ except for $y_j$. Whenever a requested point $x_t$ is not covered, $B_j$ moves the server from $x_{t-1}$ to $x_t$.

Let $S_j$, $1 \leq j \leq k$, be the set of points covered by $B_j$'s servers. We will show that throughout the execution of $\sigma$, the sets $S_j$ are pairwise different. This implies that at any step, only one of the algorithms $B_j$ has to move that thus

$$\sum_{j=1}^{k} C_{B_j}(\sigma) = \sum_{t=2}^{m} dist(x_{t-1}, x_t) = \sum_{t=1}^{m-1} dist(x_t, x_{t+1}).$$

The last sum is equal to $A$'s cost, except for the last term, which can be neglected on long request sequences.

Consider two indices $j, l$ with $1 \leq j, l \leq k$. We show by induction on the number of requests processed so far that $S_j \neq S_l$. The statement is true initially. Consider request $x_t = \sigma(t)$. If $x_t$ is in both sets, then the sets do not change. If $x_t$ is not present in one of the sets, say $B_j$, then a server is moved from $x_{t-1}$ to $x_t$. Since $x_{t-1}$ is still covered by $B_l$, the statement holds after the request. □

Manasse *et al.* also conjectured that there exists a deterministic $k$-competitive online $k$-server algorithm. Only very recently, Koutsoupias and Papadimitriou [37] showed that there is a $(2k-1)$-competitive algorithm. Before, $k$-competitive algorithms were known for special metric spaces and special values of $k$. It is worthwhile to note that the greedy algorithm, which always moves the closest server to the requested point, is not competitive.

Koutsoupias and Papadimitriou analyzed the WORK FUNCTION algorithm. Let $X$ be a configuration of the servers. Given a request sequence $\sigma = \sigma(1), \ldots, \sigma(t)$, the *work function* $w(X)$ is the minimal cost to serve $\sigma$ and end in configuration $X$.

**Algorithm WORK FUNCTION**: Suppose that the algorithm has served $\sigma = \sigma(1), \ldots, \sigma(t-1)$ and that a new request $r = \sigma(t)$

arrives. Let $X$ be the current configuration of the servers and let $x_i$ be the point where server $s_i$, $1 \leq i \leq k$, is located. Serve the request by moving the server $s_i$ that minimizes

$$w(X_i) + dist(x_i, r),$$

where $X_i = X - \{x_i\} + \{r\}$.

As mentioned above, the algorithm achieves the following performance [37].

**Theorem 8** *The WORK FUNCTION algorithm is* $(2k-1)$*-competitive in an arbitrary metric space.*

A very elegant randomized rule for moving servers was proposed by Raghavan and Snir [45].

**Algorithm HARMONIC**: Suppose that there is a new request at point $r$ and that server $s_i$, $1 \leq i \leq k$, is currently at point $x_i$. Move server $s_i$ with probability

$$p_i = \frac{1/dist(x_i, r)}{\sum_{j=1}^{k} 1/dist(x_j, r)}$$

to the request.

Intuitively, the closer a server is to the request, the higher the probability that it will be moved. Grove [32] proved that the HARMONIC algorithm has a competitive ratio of $c \leq \frac{5}{4}k \cdot 2^k - 2k$. The competitiveness of HARMONIC is not better than $k(k+1)/2$, see [43].

The main open problem in the area of the $k$-server problem is to develop randomized online algorithms that have a competitive ratio of $c < k$ in an arbitrary metric space.

# 5 The list update problem

The list update problem is to maintain a dictionary as an unsorted linear list. Consider a set of items that is represented as a linear linked list. We receive a request sequence $\sigma$, where each request is one of the following operations. (1) It can be an *access* to an item in the list, (2) it can be an *insertion* of a new item into the list, or (3) it can be a *deletion* of an item.

To access an item, a list update algorithm starts at the front of the list and searches linearly through the items until the desired item is found. To insert a new item, the algorithm first scans the entire list to verify that the item is not already present and then inserts the item at the end of the list. To delete an item, the algorithm scans the list to search for the item and then deletes it.

In serving requests a list update algorithm incurs cost. If a request is an access or a delete operation, then the incurred cost is $i$, where $i$ is the position of the requested item in the list. If the request is an insertion, then the cost is $n+1$, where $n$ is the number of items in the list before the insertion. While processing a request sequence, a list update algorithm may rearrange the list. Immediately after an access or insertion, the requested item may be moved at no extra cost to any position closer to the front of the list. These exchanges are called *free exchanges.* Using free exchanges, the algorithm can lower the cost on subsequent requests. At any time two adjacent items in the list may be exchanged at a cost of 1. These exchanges are called *paid exchanges.*

With respect to the list update problem, we require that a $c$-competitive online algorithm has a performance ratio of $c$ *for all size lists.* More precisely, a deterministic online algorithm for list update is called $c$-competitive if there is a constant $a$ such that for all size lists and all request sequences $\sigma$,

$$C_A(\sigma) \leq c \cdot C_{OPT}(\sigma) + a.$$

The competitive ratios of randomized online algorithms are defined similarly.

Linear lists are one possibility to represent a dictionary. Certainly, there are other data structures such as balanced search trees or hash tables that, depending on the given application, can maintain a dictionary in a more efficient way. In general, linear lists are useful when the dictionary is small and consists of only a few dozen items [19]. Furthermore, list update algorithms have been used as subroutines in algorithms for computing point maxima and convex hulls [18, 30]. Recently, list update techniques have been very successfully applied in the development of data compression algorithms [24]. We discuss this application in detail in Section 6.

## 5.1 Deterministic online algorithms

There are three well-known deterministic online algorithms for the list update problem.

- **Move-To-Front**: Move the requested item to the front of the list.
- **Transpose**: Exchange the requested item with the immediately preceding item in the list.
- **Frequency-Count**: Maintain a frequency count for each item in the list. Whenever an item is requested, increase its count by 1. Maintain the list so that the items always occur in nonincreasing order of frequency count.

The formulations of list update algorithms generally assume that a request sequence consists of accesses only. It is obvious how to extend the algorithms so that they can also handle insertions and deletions. On an insertion, the algorithm first appends the new item at the end of the list and then executes the same steps as if the item was requested for the first time. On a deletion, the algorithm first searches for the item and then just removes it.

In the following, we discuss the algorithms Move-To-Front, Transpose and Frequency-Count. We note that Move-To-Front and Transpose are *memoryless* strategies, i.e., they do not need any extra memory to decide where a requested item should be moved. Thus, from a practical point of view, they are more attractive than Frequency-Count. Sleator and Tarjan [48] analyzed the competitive ratios of the three algorithms.

**Theorem 9** *The Move-To-Front algorithm is 2-competitive.*

**Proof:** Consider a request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ of length $m$. First suppose that $\sigma$ consists of accesses only. We will compare simultaneous runs of Move-To-Front and OPT on $\sigma$ and evaluate online and offline cost using a potential function $\Phi$.

The potential function we use is the number of inversions in Move-To-Front's list with respect to OPT's list. An *inversion* is a pair $x, y$ of items such that $x$ occurs before $y$ Move-To-Front's list and after $y$ in OPT's list. We assume without loss of generality that Move-To-Front and OPT start with the same list so that the initial potential is 0.

For any $t$, $1 \le t \le m$, let $C_{MTF}(t)$ and $C_{OPT}(t)$ denote the actual cost incurred by Move-To-Front and OPT in serving $\sigma(t)$. Furthermore, let $\Phi(t)$ denote the potential after $\sigma(t)$ is served. The *amortized cost* incurred by Move-To-Front on $\sigma(t)$ is defined as $C_{MTF}(t) + \Phi(t) - \Phi(t-1)$. We will show that for any $t$,

$$C_{MTF}(t) + \Phi(t) - \Phi(t-1) \le 2C_{OPT}(t) - 1. \qquad (5)$$

Summing this expression for all $t$ we obtain $\sum_{t=1}^{m} C_{MTF}(t) + \Phi(m) - \Phi(0) \le \sum_{t=1}^{m} 2C_{OPT}(t) - m$, i.e., $C_{MTF}(\sigma) \le 2C_{OPT}(\sigma) - m + \Phi(0) -$

$\Phi(m)$. Since the initial potential is 0 and the final potential is non-negative, the theorem follows.

In the following we will show inequality (5) for an arbitrary $t$. Let $x$ be the item requested by $\sigma(t)$. Let $k$ denote the number of items that precede $x$ in Move-To-Front's and OPT's list. Furthermore, let $l$ denote the number of items that precede $x$ in Move-To-Front's list but follow $x$ in OPT's list. We have $C_{MTF}(t) = k + l + 1$ and $C_{OPT}(t) \geq k + 1$.

When Move-To-Front serves $\sigma(t)$ and moves $x$ to the front of the list, $l$ inversions are destroyed and at most $k$ new inversions are created. Thus

$$\begin{aligned} C_{MTF}(t) + \Phi(t) - \Phi(t-1) &\leq C_{MTF}(t) + k - l \;=\; 2k + 1 \\ &\leq 2C_{OPT}(t) - 1. \end{aligned}$$

Any paid exchange made by OPT when serving $\sigma(t)$ can increase the potential by 1, but OPT also pays 1. We conclude that inequality (5) holds.

The above arguments can be extended easily to analyze an insertion or deletion. On an insertion, $C_{MTF}(t) = C_{OPT}(t) = n + 1$, where $n$ is the number of items in the list before the insertion, and at most $n$ new inversions are created. On a deletion, $l$ inversions are removed and no new inversion is created. □

Sleator and Tarjan [48] showed that, in terms of competitiveness, Move-To-Front is superior to Transpose and Frequency-Count.

**Proposition 1** *The algorithms Transpose and Frequency-Count are not c-competitive for any constant c.*

Recently, Albers [2] presented another deterministic online algorithm for the list update problem. The algorithm belongs to the Timestamp($p$) family of algorithms that were introduced in the context of randomized online algorithms and that are defined for any real number $p \in [0, 1]$. For $p = 0$, the algorithm is deterministic and can be formulated as follows.

**Algorithm Timestamp(0)**: Insert the requested item, say $x$, in front of the first item in the list that precedes $x$ and that has been requested at most once since the last request to $x$. If there is no such item or if $x$ has not been requested so far, then leave the position of $x$ unchanged.

**Theorem 10** *The Timestamp(0) algorithm is 2-competitive.*

Note that Timestamp(0) is not memoryless. We need information on past requests in order to determine where a requested item should be moved. In fact, in the most straightforward implementation of the algorithm we need a second pass through the list to find the position where the accessed item must be inserted. Timestamp(0) is interesting because it has a better overall performance than Move-To-Front. The algorithm achieves a competitive ratio of 2, as does Move-To-Front. However, as we shall see in Section 5.3, Timestamp(0) is considerably better than Move-To-Front on request sequences that are generated by probability distributions.

Karp and Raghavan [39] developed a lower bound on the competitiveness that can be achieved by deterministic online algorithms. This lower bound implies that Move-To-Front and Timestamp(0) have an optimal competitive ratio.

**Theorem 11** *Let $A$ be a deterministic online algorithm for the list update algorithm. If $A$ is $c$-competitive, then $c \geq 2$.*

**Proof:** Consider a list of $n$ items. We construct a request sequence that consist of accesses only. Each request is made to the item that is stored at the last position in $A$'s list. On a request sequence $\sigma$ of length $m$ generated in this way, $A$ incurs a cost of $C_A(\sigma) = mn$. Let OPT' be the optimum static offline algorithm. OPT' first sorts the items in the list in order of nonincreasing request frequencies and then serves $\sigma$ without making any further exchanges. When rearranging the list, OPT' incurs a cost of at most $n(n-1)/2$. Then the requests in $\sigma$ can be served at a cost of at most $m(n+1)/2$. Thus $C_{OPT}(\sigma) \le m(n+1)/2 + n(n-1)/2$. For long request sequences, the additive term of $n(n-1)/2$ can be neglected and we obtain

$$C_A(\sigma) \ge \tfrac{2n}{n+1} \cdot C_{OPT}(\sigma).$$

The theorem follows because the competitive ratio must hold for all list lengths. □

## 5.2 Randomized online algorithms

We analyze randomized online algorithms for the list update problem against oblivious adversaries. It was shown by Reingold *et al.* [46] that against adaptive online adversaries, no randomized online algorithm for list update can be better than 2-competitive. Recall that, by Theorem 3, a lower bound of 2 also holds against adaptive offline adversaries.

Many randomized online algorithms for list update have been proposed in the literature [34, 35, 46, 2, 5]. We present the two most important algorithms. Reingold *et al.* [46] gave a very simple algorithm, called BIT.

**Algorithm Bit**: Each item in the list maintains a bit that is complemented whenever the item is accessed. If an access causes a bit to change to 1, then the requested item is moved to the front of the list. Otherwise the list remains unchanged. The bits of the items are initialized independently and uniformly at random.

**Theorem 12** *The Bit algorithm is 1.75-competitive against any oblivious adversary.*

Reingold *et al.* analyzed Bit using an elegant modification of the potential function given in the proof of Theorem 9. Again, an inversion is a pair of items $x, y$ such that $x$ occurs before $y$ in Bit's list and after $y$ in OPT's list. An inversion has *type 1* if $y$'s bit is 0 and *type 2* if $y$'s bit is 1. Now, the potential is defined as the number of type 1 inversions plus twice the number of type 2 inversions.

Interestingly, it is possible to combine the algorithms Bit and Timestamp(0), see Albers *et al.* [5]. This combined algorithm achieves the best competitive ratio that is currently known for the list update problem.

**Algorithm Combination**: With probability 4/5 the algorithm serves a request sequence using Bit, and with probability 1/5 it serves a request sequence using Timestamp(0).

**Theorem 13** *The algorithm Combination is 1.6-competitive against any oblivious adversary.*

**Proof:** The analysis consists of two parts. In the first part we show that given any request sequence $\sigma$, the cost incurred by Combination and OPT can be divided into costs that are caused by each unordered pair $\{x, y\}$ of items $x$ and $y$. Then, in the second part, we compare online and offline cost for each pair $\{x, y\}$. This method of analyzing cost by considering pairs of items was first introduced by Bentley and McGeoch [19] and later used in [2, 34]. In the following we always assume that serving a request to the $i$-th item in the list incurs a cost of $i-1$ rather than $i$. Clearly, if Combination is 1.6-competitive in this $i-1$ cost model, it is also 1.6-competitive in the $i$-cost model.

Let $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ be an arbitrary request sequence of length $m$. For the reduction to pairs we need some notation. Let $S$ be the set of items in the list. Consider any list update algorithm $A$ that processes $\sigma$. For any $t \in [1, m]$ and any item $x \in S$, let $C_A(t, x)$

be the cost incurred by item $x$ when $A$ serves $\sigma(t)$. More precisely, $C_A(t,x) = 1$ if item $x$ precedes the item requested by $\sigma(t)$ in $A$'s list at time $t$; otherwise $C_A(t,x) = 0$. If $A$ does not use paid exchanges, then the total cost $C_A(\sigma)$ incurred by $A$ on $\sigma$ can be written as

$$\begin{aligned}
C_A(\sigma) &= \sum_{t\in[1,m]} \sum_{x\in S} C_A(t,x) = \sum_{x\in S} \sum_{t\in[1,m]} C_A(t,x) \\
&= \sum_{x\in S} \sum_{y\in S} \sum_{\substack{t\in[1,m]\\ \sigma(t)=y}} C_A(t,x) \\
&= \sum_{\substack{\{x,y\}\\ x\neq y}} \Big( \sum_{\substack{t\in[1,m]\\ \sigma(t)=x}} C_A(t,y) + \sum_{\substack{t\in[1,m]\\ \sigma(t)=y}} C_A(t,x) \Big).
\end{aligned}$$

For any unordered pair $\{x,y\}$ of items $x \neq y$, let $\sigma_{xy}$ be the request sequence that is obtained from $\sigma$ if we delete all requests that are neither to $x$ nor to $y$. Let $C_{BIT}(\sigma_{xy})$ and $C_{TS}(\sigma_{xy})$ denote the costs that Bit and Timestamp(0) incur in serving $\sigma_{xy}$ on a two item list that consist of only $x$ and $y$. Obviously, if Bit serves $\sigma$ on the long list, then the relative position of $x$ and $y$ changes in the same way as if Bit serves $\sigma_{xy}$ on the two item list. The same property holds for Timestamp(0). This follows from Lemma 2, which can easily be shown by induction on the number of requests processed so far.

**Lemma 2** *At any time during the processing of $\sigma$, $x$ precedes $y$ in Timestamp(0)'s list if and only if one of the following statements holds: (a) the last requests made to $x$ and $y$ are of the form $xx$, $xyx$ or $xxy$; (b) $x$ preceded $y$ initially and $y$ was requested at most once so far.*

Thus, for algorithm $A \in \{\text{Bit, Timestamp}(0)\}$ we have

$$C_A(\sigma_{xy}) = \sum_{\substack{t\in[1,m]\\ \sigma(t)=x}} C_A(t,y) + \sum_{\substack{t\in[1,m]\\ \sigma(t)=y}} C_A(t,x)$$

$$C_A(\sigma) = \sum_{\substack{\{x,y\}\\ x\neq y}} C_A(\sigma_{xy}). \tag{6}$$

Note that Bit and Timestamp(0) do not incur paid exchanges. For the optimal offline cost we have

$$C_{OPT}(\sigma_{xy}) \leq \sum_{\substack{t\in[1,m]\\ \sigma(t)=x}} C_{OPT}(t,y) + \sum_{\substack{t\in[1,m]\\ \sigma(t)=y}} C_{OPT}(t,x) \; + \; p(x,y)$$

$$C_{OPT}(\sigma) \geq \sum_{\substack{\{x,y\}\\ x\neq y}} C_{OPT}(\sigma_{xy}), \tag{7}$$

where $p(x,y)$ denotes the number of paid exchanges incurred by OPT in moving $x$ in front of $y$ or $y$ in front of $x$. Here, only inequality signs hold because if OPT serves $\sigma_{xy}$ on the two item list, then it can always arrange $x$ and $y$ optimally in the list, which might not be possible if OPT serves $\sigma$ on the entire list. Note that the expected cost $E[C_{CB}(\sigma_{xy})]$ incurred by Combination on $\sigma_{xy}$ is

$$E[C_{CB}(\sigma_{xy})] = \frac{4}{5}E[C_{BIT}(\sigma_{xy})] + \frac{1}{5}E[C_{TS}(\sigma_{xy})]. \tag{8}$$

In the following we will show that for any pair $\{x,y\}$ of items $E[C_{CB}(\sigma_{xy})] \leq 1.6C_{OPT}(\sigma_{xy})$. Summing this inequality for all pairs $\{x,y\}$, we obtain, by equations (6),(7) and (8), that Combination is 1.6-competitive.

Consider a fixed pair $\{x,y\}$ with $x\neq y$. We partition the request sequence $\sigma_{xy}$ into phase. The first phase starts with the first request in $\sigma_{xy}$ and ends when, for the first time, there are two requests to the same item and the next request is different. The second phase starts with that next request and ends in the same way as the first phase. The third and all remaining phases are constructed in the same way as the second phase. The phases we obtain are of the following types: $x^k$ for some $k\geq 2$; $(xy)^kx^l$ for some $k\geq 1, l\geq 2$; $(xy)^ky^l$ for some $k\geq 1$, $l\geq 1$. Symmetrically, we have $y^k$, $(yx)^ky^l$ and $(yx)^kx^l$.

Since a phase ends with (at least) two requests to the same item, the item requested last in the phase precedes the other item in the two item list maintained by Bit and Timestamp(0). Thus the item requested first in a phase is always second in the list. Without loss of generality we can assume the same holds for OPT, because when OPT serves two consecutive requests to the same item, it cannot cost more to move that item to the front of the two item list after the first request. The expected cost incurred by Bit, Timestamp(0) (denoted by TS(0)) and OPT are given in the table below. The symmetric phases with $x$ and $y$ interchanged are omitted. We assume without generality that at the beginning of $\sigma_{xy}$, $y$ precedes $x$ in the list.

| Phase | $Bit$ | $TS(0)$ | $OPT$ |
|---|---|---|---|
| $x^k$ | $\frac{3}{2}$ | $2$ | $1$ |
| $(xy)^k x^l$ | $\frac{3}{2}k+1$ | $2k$ | $k+1$ |
| $(xy)^k y^l$ | $\frac{3}{2}k+\frac{1}{4}$ | $2k-1$ | $k$ |

The entries for OPT are obvious. When Timestamp(0) serves a phase $(xy)^k x^l$, then the first two request $xy$ incur a cost of 1 and 0, respectively, because $x$ is left behind $y$ on the first request to $x$. On all subsequent requests in the phase, the requested item is always moved to the front of the list. Therefore, the total cost on the phase is $1+0+2(k-1)+1=2k$. Similarly, Timestamp(0) serves $(xy)^k y^l$ with cost $2k-1$.

For the analysis of Bit's cost we need two lemmata.

**Lemma 3** *For any item $x$ and any $t \in [1,m]$, after the $t$-th request in $\sigma$, the value of $x$'s bit is equally likely to be 0 or 1, and the value is independent of the bits of the other items.*

**Lemma 4** *Suppose that Bit has served three consecutive requests $yxy$ in $\sigma_{xy}$, or two consecutive requests $xy$ where initially $y$ preceded $x$. Then $y$ is in front of $x$ with probability $\frac{3}{4}$. The analogous statement holds when the roles of $x$ and $y$ are interchanged.*

Clearly, the expected cost spent by Bit on a phase $x^k$ is $1 + \frac{1}{2} + 0(k-2) = \frac{3}{2}$. Consider a phase $(xy)^k x^l$. The first two requests $xy$ incur a expected cost of 1 and $\frac{1}{2}$, respectively. By Lemma 4, each remaining request in the string $(xy)^k$ and the first request in $x^l$ have an expected cost of $\frac{3}{4}$. Also by Lemma 4, the second request in $x^l$ costs $1 - \frac{3}{4} = \frac{1}{4}$. All other requests in $x^l$ are free. Therefore, Bit pays an expected cost of $1 + \frac{1}{2} + \frac{3}{2}(k-1) + \frac{3}{4} + \frac{1}{4} = \frac{3}{2}k + 1$ on the phase. Similarly, we can evaluate a phase $(xy)^k y^l$.

The Combination algorithm serves a request sequence with probability $\frac{4}{5}$ using Bit and with probability $\frac{1}{5}$ using Timestamp(0). Thus, by the above table, Combination has an expected cost of 1.6 on a phase $x^k$, a cost of $1.6k + 0.8$ on a phase $(xy)^k x^l$, and a cost $1.6k$ on a phase $(xy)^k y^l$. In each case this is at most 1.6 times the cost of OPT.

In the above proof we assume that a request sequence consist of accesses only. However, the analysis is easily extended to the case that insertions and deletions occur, too. For any item $x$, consider the time intervals during which $x$ is contained in the list. For each of these intervals, we analyze the cost caused by any pair $\{x, y\}$, where $y$ is an item that is (temporarily) present during the interval.
$\square$ $\square$

Teia [49] presented a lower bound for randomized list update algorithms.

**Theorem 14** *Let A be a randomized online algorithm for the list update problem. If A is c-competitive against any oblivious adversary, then* $c \geq 1.5$.

An interesting open problem is to give tight bounds on the competitive ratio that can be achieved by randomized online algorithms against oblivious adversaries.

## 5.3 Average case analyses of list update algorithms

In this section we study a restricted class of request sequences: request sequences that are generated by a probability distribution. Consider a list of $n$ items $x_1, x_2, \ldots, x_n$, and let $\vec{p} = (p_1, p_2, \ldots, p_n)$ be a vector of positive probabilities $p_i$ with $\sum_{i=1}^{n} p_i = 1$. We study request sequences that consist of accesses only, where each request it made to item $x_i$ with probability $p_i$, $1 \le i \le n$. It is convenient to assume that $p_1 \ge p_2 \ge \cdots \ge p_n$.

There are many results known on the performance of list update algorithms when a request sequence is generated by a probability distribution, i.e. by a discrete memoryless source. In fact, the algorithms Move-To-Front, Transpose and Frequency-Count given in Section 5.1 as well as their variants were proposed as heuristics for these particular request sequences.

We are now interested in the asymptotic expected cost incurred by a list update algorithm. For any algorithm $A$, let $E_A(\vec{p})$ denote the asymptotic expected cost incurred by $A$ in serving a single request in a request sequence generated by the distribution $\vec{p} = (p_1, \ldots, p_n)$. In this situation, the performance of an online algorithm has generally been compared to that of the *optimal static ordering*, which we call STAT. The optimal static ordering first arranges the items $x_i$ in nonincreasing order by probabilities and then serves a request sequence without changing the relative position of items. Clearly, $E_{STAT}(\vec{p}) = \sum_{i=1}^{n} i p_i$ for any distribution $\vec{p} = (p_1, \ldots, p_n)$.

We first study the algorithms Move-To-Front(MTF), Transpose(T) and Frequency-Count(FC). By the strong law of large numbers we have $E_{FC}(\vec{p}) = E_{STAT}(\vec{p})$ for any probability distribution $\vec{p}$ [47]. However, as mentioned in Section 5.1, Frequency-Count may need a large amount of extra memory to serve a request sequence.

Chung *et al.* [25] gave an upper bound of Move-To-Front's performance.

**Theorem 15** *For any probability distribution $\vec{p}$,*

$$E_{MTF}(\vec{p}) \le \frac{\pi}{2} E_{STAT}(\vec{p}).$$

This bound is tight as was shown by Gonnet *et al.* [28].

**Theorem 16** *For any $\epsilon > 0$, there exists a probability distribution $\vec{p}_\epsilon$ with*

$$E_{MTF}(\vec{p}_\epsilon) \ge (\frac{\pi}{2} - \epsilon) E_{STAT}(\vec{p}_\epsilon).$$

Rivest [47] proved that Transpose performs better than Move-To-Front on distributions.

**Theorem 17** *For any distribution $\vec{p} = (p_1, \ldots, p_n)$,*

$$E_T(\vec{p}) \le E_{MTF}(\vec{p}).$$

*The inequality is strict unless $n = 2$ or $p_i = 1/n$ for $i = 1, \ldots, n$.*

Finally, we consider the Timestamp(0) algorithm that was also presented in Section 5.1. It was shown in [4] that Timestamp(0) has a better performance then Move-To-Front if request sequences are generated by probability distributions. Let $E_{TS}(\vec{p})$ denote the asymptotic expected cost incurred by Timestamp(0).

**Theorem 18** *For any probability distribution $\vec{p}$,*

$$E_{TS}(\vec{p}) \le 1.34 E_{STAT}(\vec{p}).$$

**Theorem 19** *For any probability distribution $\vec{p}$,*

$$E_{TS}(\vec{p}) \le 1.5 E_{OPT}(\vec{p}).$$

Note that $E_{OPT}(\vec{p})$ is the asymptotic expected cost incurred by the optimal offline algorithm OPT, which may dynamically rearrange the list while serving a request sequence. Thus, this algorithm is much stronger than STAT. The algorithm Timestamp(0) is the only algorithm whose asymptotic expected cost has been compared to $E_{OPT}(\vec{p})$.

The bound given in Theorem 19 holds with high probability. More precisely, for every distribution $\vec{p} = (p_1, \ldots, p_n)$, and $\epsilon > 0$, there exist constants $c_1, c_2$ and $m_0$ dependent on $\vec{p}, n$ and $\epsilon$ such that for any request sequence $\sigma$ of length $m \geq m_0$ generated by $\vec{p}$,

$$Prob\{C_{TS}(\sigma) > (1.5 + \epsilon)C_{OPT}(\sigma)\} \leq c_1 e^{-c_2 m}.$$

# 6 Data compression based on linear lists

Linear lists can be used to build locally adaptive data compression schemes. This application of linear lists recently became of considerable importance, due to a paper by Burrows and Wheeler. In [24], Burrows and Wheeler developed a data compression scheme using unsorted lists that achieves a better compression than Ziv-Lempel based algorithms. Before describing their algorithm, we first present a data compression scheme given by Bentley *et al.* [20] and discuss theoretical as well as experimental results.

## 6.1 Theoretical results

In data compression we are given a string $S$ that shall be *compressed*, i.e., that shall be represented using fewer bits. The string $S$ consists of *symbols*, where each symbol is an element of the alphabet $\Sigma = \{x_1, \ldots, x_n\}$. The idea of data compression schemes using linear lists it to convert the string $S$ of symbols into a string $I$ of integers. An *encoder* maintains a linear list of symbols contained in $\Sigma$ and reads the symbols in the string $S$. Whenever the symbol $x_i$ has to be compressed, the encoder looks up the current position of $x_i$ in the linear list, outputs this position and updates the list using a list update rule. If symbols to be compressed are moved closer to the front of the list, then frequently occurring symbols can be encoded with small integers.

A *decoder* that receives $I$ and has to recover the original string $S$ also maintains a linear list of symbols. For each integer $j$ it reads from $I$, it looks up the symbol that is currently stored at position $j$. Then the decoder updates the list using the same list update rule as the encoder. Clearly, when the string $I$ is actually stored or transmitted, each integer in the string should be coded again using a variable length prefix code.

In order to analyze the above data compression scheme one has to specify how an integer $j$ in $I$ shall be encoded. Elias [26] presented several coding schemes that encode an integer $j$ with essentially $\log j$ bits. The simplest version of his schemes encodes $j$ with $1 + 2\lfloor \log j \rfloor$ bits. The code for $j$ consists of a prefix of $\lfloor \log j \rfloor$ 0's followed by the binary representation of $j$, which requires $1 + \lfloor \log j \rfloor$ bits. A second encoding scheme is obtained if the prefix of $\lfloor \log j \rfloor$ 0's followed by the first 1 in the binary representation of $j$ is coded again using this simple scheme. Thus, the second code uses $1 + \lfloor \log j \rfloor + 2\lfloor \log(1 + \log j) \rfloor$ bits to encode $j$.

Bentley *et al.* [20] analyzed the above data compression algorithm if encoder and decoder use Move-To-Front as list update rule. They assume that an integer $j$ is encoded with $f(j) = 1 + \lfloor \log j \rfloor + 2\lfloor \log(1 + \log j) \rfloor$ bits. For a string $S$, let $A_{MTF}(S)$ denote the average number of bits needed by the compression algorithm to encode one symbol in $S$. Let $m$ denote the length of $S$ and let $m_i$, $1 \le i \le n$, denote the number of occurrences of the symbol $x_i$ in $S$.

**Theorem 20** *For any input sequence $S$,*

$$A_{MTF}(S) \leq 1 + H(S) + 2\log(1 + H(S)),$$

*where* $H(S) = \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i})$.

The expression $H(S) = \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i})$ is the "empirical entropy" of $S$. The empirical entropy is interesting because it corresponds to the average number of bits per symbol used by the optimal static Huffman encoding for a sequence. Thus, Theorem 20 implies that Move-To-Front based encoding is almost as good as static Huffman encoding.

**Proof of Theorem 20:** We assume without loss of generality that the encoder starts with an empty linear list and inserts new symbols as they occur in the string $S$. Let $f(j) = 1 + \lfloor \log j \rfloor + 2\lfloor \log(1 + \log j) \rfloor$. Consider a fixed symbol $x_i$, $1 \leq i \leq n$, and let $q_1, q_2, \ldots, q_{m_i}$ be the positions at which the symbol $x_i$ occurs in the string $S$. The first occurrence of $x_i$ in $S$ can the encoded with $f(q_1)$ bits and the $k$-th occurrence of $x_i$ can be encoded with $f(q_k - q_{k-1})$ bits. The $m_i$ occurrences of $x_i$ can be encoded with a total of

$$f(q_1) + \sum_{k=1}^{m_i} f(q_k - q_{k-1})$$

bits. Note that $f$ is a concave function. We now apply Jensen's inequality, which states that for any concave function $f$ and any set $\{w_1, \ldots, w_n\}$ of positive reals whose sum is 1, $\sum_{i=1}^{n} w_i f(y_i) \leq f(\sum_{i=1}^{n} w_i y_i)$ [33]. Thus, the $m_i$ occurrences of $x_i$ can be encoded with at most

$$m_i f(\frac{1}{m_i}(q_1 + \sum_{k=2}^{m_i}(q_k - q_{k-1})) = m_i f(\frac{q_{m_i}}{m_i}) \leq m_i f(\frac{m}{m_i})$$

bits. Summing the last expression for all symbols $x_i$ and dividing by $m$, we obtain

$$A_{MTF}(S) = \sum_{i=1}^{n} \frac{m_i}{m} f(\frac{m}{m_i}).$$

The definition of $f$ gives

$$\begin{aligned} A_{MTF}(S) &\leq \sum_{i=1}^{n} \frac{m_i}{m} + \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i}) + \sum_{i=1}^{n} \frac{m_i}{m} 2\log(1+\log(\frac{m}{m_i})) \\ &\leq \sum_{i=1}^{n} \frac{m_i}{m} + \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i}) + 2\log(\sum_{i=1}^{n} \frac{m_i}{m} + \sum_{i=1}^{n} \frac{m_i}{m} \log(\frac{m}{m_i})) \\ &= 1 + H(S) + 2\log(1+H(S)). \end{aligned}$$

The second inequality follows again from Jensen's inequality. □

Bentley *et al.* [20] also considered strings that are generated by probability distributions, i.e., by discrete memoryless sources $\vec{p} = (p_1, \ldots, p_n)$. The $p_i$'s are positive probabilities that sum to 1. In a string $S$ generated by $\vec{p} = (p_1, \ldots, p_n)$, each symbol is equal to $x_i$ with probability $p_i$, $1 \leq i \leq n$. Let $B_{MTF}(\vec{p})$ denote the expected number of bits needed by Move-To-Front to encode one symbol in a string generated by $\vec{p} = (p_1. \ldots, p_n)$.

**Theorem 21** *For any $\vec{p} = (p_1, \ldots, p_n)$,*

$$B_{MTF}(\vec{p}) \leq 1 + H(\vec{p}) + 2\log(1+H(\vec{p})),$$

*where $H(\vec{p}) = \sum_{i=1}^{n} p_i \log(1/p_i)$ is the entropy of the source.*

Shannon's source coding theorem (see e.g. Gallager [32]) implies that the number $B_{MTF}(\vec{p})$ of bits needed by Move-To-Front encoding is optimal, up to a constant factor.

Albers and Mitzenmacher [4] analyzed the data compression algorithm if encoder and decoder use Timestamp(0) as list update algorithm. They showed that a statement analogous to Theorem 20 holds. More precisely, for any string $S$, let $A_{MTF}(S)$ denote the average number of bits needed by Timestamp(0) to encode one symbol in $S$. Then, $A_{TS}(S) \leq 1 + H(S) + 2\log(1+H(S))$, where $H(S)$ is the empirical entropy of $S$. For strings generated by discrete memoryless sources, Timestamp(0) achieves a better compression than Move-To-Front.

**Theorem 22** *For any* $\vec{p} = (p_1, p_2, \ldots, p_n)$,

$$B_{TS}(\vec{p}) \leq 1 + \overline{H}(\vec{p}) + 2\log(1 + \overline{H}(\vec{p})),$$

*where* $\overline{H}(\vec{p}) = \sum_{i=1}^{n} p_i \log(1/p_i) + \log(1 - \sum_{i \leq j} p_i p_j (p_i - p_j)^2/(p_i + p_j)^2)$.

Note that $0 \leq \sum_{i \leq j} p_i p_j (p_i - p_j)^2/(p_i + p_j)^2 < 1$.

## 6.2 Experimental results

The above data compression algorithm, based on Move-To-Front or Timestamp(0), was analyzed experimentally [4, 20]. In general, the algorithm can be implemented in two ways. In a *byte-level* scheme, each ASCII character in the input string is regarded as a symbol that is encoded individually. In contrast, in a *word-level* scheme each word, i.e. each longest sequence of alphanumeric and nonalphanumeric characters, represents a symbol. Albers and Mitzenmacher [4] compared Move-To-Front and Timestamp(0) based encoding on the Calgary Compression Corpus [52], which consists of files commonly used to evaluate data compression algorithms. In the byte-level implementations, Timestamp(0) achieves a better compression than Move-To-Front. The improvement is typically 6–8%. However, the byte-level schemes perform far worse than standard UNIX utilities such as pack or compress. In the word-level implementations, the compression achieved by Move-To-Front and Timestamp(0) is comparable to that of the UNIX utilities. However, in this situation, the improvement achieved by Timestamp(0) over Move-To-Front is only about 1%.

Bentley *et al.* [20] implemented a word-level scheme based on Move-To-Front that uses a linear list of limited size. Whenever the encoder reads a word from the input string that is not contained in the list, the word is written in non-coded form onto the output string. The word is inserted as new item at the front of the list and, if the current list length exceeds the allowed length, the last item of the list is deleted. Such a list acts like a cache. Bentley *et al.* tested the compression scheme with various list lengths on several text and Pascal files. If the list may contain up to 256 items, the compression achieved is comparable to that of word-based Huffman encoding and sometimes better.

## 6.3 The compression algorithm by Burrows and Wheeler

As mentioned in the beginning of this section, Burrows and Wheeler [24] developed a very effective data compression algorithm using self-organizing lists that achieves a better compression than Ziv-Lempel based schemes. The algorithm by Burrows and Wheeler first applies a reversible transformation to the string $S$. The purpose of this transformation is to group together instances of a symbol $x_i$ occurring in $S$. The resulting string $S'$ is then encoded using the Move-To-Front algorithm.

More precisely, the transformed string $S'$ is computed as follows. Let $m$ be the length of $S$. The algorithm first computes the $m$ rotations (cyclic shifts) of $S$ and sorts them lexicographically. Then it extracts the last character of these rotations. The $k$-th symbol of $S'$ is the last symbol of the $k$-th sorted rotation. The algorithm also computes the index $J$ of the original string $S$ in the sorted list of rotations. Burrows and Wheeler gave an efficient algorithm to compute the original string $S$ given only $S'$ and $J$.

In the sorting step, rotations that start with the same symbol are grouped together. Note that in each rotation, the initial symbol is adjacent to the final symbol in the original string $S$. If in the string $S$, a symbol $x_i$ is very often followed by $x_j$, then the occurrences of $x_j$ are grouped together in $S'$. For this reason, $S'$ generally has a very high locality of reference and can be encoded very effectively with Move-To-Front. The paper by Burrows and Wheeler gives a very detailed description of the algorithm and reports of experimental results. On the Calgary Compression Corpus, the algorithm outperforms the UNIX utilities compress and gzip and the improvement is 13% and 6%, respectively.

# 7 Distributed data management

Consider a network of processors, each of which has its local memory. A global shared memory is modeled by distributing the physical pages among the local memories. Accesses to the global memory are accomplished by accessing the local memories. Suppose a processor $p$ wants to read a memory address from page $B$. If $B$ is stored in $p$'s local memory, then this read operation can be executed locally. Otherwise, $p$ determines a processor $q$ holding the page and sends a request to $q$. The desired information is then transmitted from $q$ to $p$, and the communication cost incurred thereby is proportional to the distance from $q$ to $p$. If $p$ has to access page $B$ frequently, it may be worthwhile to move or copy $B$ from $q$ to $p$ because subsequent accesses will become cheaper. However, transmitting an entire page incurs a high communication cost proportional to the page size times the distance from $q$ to $p$.

If a page is writable, it is reasonable to store only one copy of the page in the entire system. This avoids the problem of keeping multiple copies of the page consistent. The *migration problem* is to decide in which local memory the single copy of the writable page should be stored so that a sequence of memory accesses can be processed at low cost. On the other hand, if a page is read-only, it is possible to keep several copies of the page in the system, i.e., a page may be copied from one local memory to another. In the *replication problem* we have to determine which local memories should contain copies of the read-only page. Finding efficient migration and replication strategies is an important problem that has been studied from a practical and theoretical point of view. In this section we study on-line algorithms for page migration and replication.

## 7.1 Formal definition of migration and replication problems

Formally, the page migration and replication problems can be described as follows. We are given an undirected graph $G$. Each node in $G$ corresponds to a processor and the edges represent the interconnection network. Associated with each edge is a *length* that is equal to the distance between the connected processors. We assume that the edge lengths satisfy the triangle inequality.

In page migration and replication we generally concentrate on one particular page. We say that *a node $v$ has the page* if the page is contained in $v$'s local memory. *A request at a node $v$* occurs if $v$ wants to read or write an address from the page. The request can be satisfied at zero cost if $v$ has the page. Otherwise the request is served by accessing a node $w$ holding the page and the incurred cost equals the distance from $v$ to $w$. After the request is satisfied, the page may be migrated or replicated from node $w$ to any other node $v'$ that does not hold the page (node $v'$ may coincide with node $v$). The cost incurred by this migration or replication is $d$ times the distance from $w$ to $v'$. Here $d$ denotes the page size factor. In practical applications, $d$ is a large value, usually several hundred or thousand. (The page may only be migrated or replicated *after* a request because it is impossible to delay the service of the memory access while the entire page is transmitted.)

A page migration or replication algorithm is usually presented with an entire sequence of requests that must be served with low total cost. An algorithm is *on-line* if it serves every request without knowledge of any future requests.

In these notes we only consider *centralized* migration and replication algorithms, i.e., each node always knows where the closest node holding the page is located in the network.

## 7.2 Page migration

Recall that in the migration problem we have to decide where the single copy of a page should reside in the network over time. There are deterministic online migration algorithms that achieve competitive ratios of 7 and 4.1, respectively, see [10, 14]. We describe an elegant randomized algorithm due to Westbrook [51].

**Algorithm COUNTER**: The algorithm maintains a global counter $C$ that takes integer values in $[0, k]$, for some positive integer $k$ to be specified later. Counter $C$ is initialized uniformly at random to an integer in $[1, k]$. On each request, $C$ is decremented by 1. If $C = 0$ after the service of the request, then the page is moved to the requesting node and $C$ is reset to $k$.

**Theorem 23** *The COUNTER algorithm is c-competitive, where* $c = \max\{2 + \frac{2d}{k}, 1 + \frac{k+1}{2d}\}$.

Westbrook showed that the best value for $k$ is $d+\frac{1}{2}(\sqrt{20d^2 - 4d + 1} - 1)$. As $d$ increases, the best competitive ratio decreases and tends to $1 + \Phi$, where $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.62$ is the Golden Ratio.

**Proof:** Let $\sigma = \sigma(1), \ldots, \sigma(m)$ be an arbitrary request sequence. We analyze the COUNTER algorithm using a potential function $\Phi$ and show that

$$E[C_{CT}(t) + \Phi(t) - \Phi(t-1)] \leq c \cdot C_{OPT}(t),$$

where $c$ is the value specified above. Here $C_{CT}(t)$ denotes the actual cost incurred by the COUNTER algorithm on request $\sigma(t)$.

We classify the actions that can occur during the processing of $\sigma(t)$ into two types of events.

I: COUNTER and OPT serve the request. This event may involve COUNTER moving the page.

II: OPT moves the page.

Let $u_C$ be the node where COUNTER has the page and let $u_{OPT}$ be the node where OPT has the page. Let

$$\Phi = (d + C)dist(u_C, u_{OPT}),$$

where $C$ is the value of the global counter maintained by the online algorithm.

We first analyze event I. Let $v$ be the node that issues the request. OPT incurs a cost of $l_0 = dist(v, u_{OPT})$. Let $l_1 = dist(u_C, u_{OPT})$ and $l_2 = dist(v, u_C)$. The actual cost incurred by COUNTER is $l_2$. Since the global counter decreases by 1, the potential decreases by $l_1$, and the amortized cost incurred by COUNTER is $l_2 - l_1$. By the triangle inequality, $l_2 - l_1 \leq l_0$.

With probability $\frac{1}{k}$, $C = 1$ before the request and thus $C = 0$ after the request. The cost of moving the page is $d \cdot l_2$. The potential before the move is $d \cdot l_1$ and after the move $(d+k)l_0$ since the global counter is reset to $k$. Thus the amortized cost of moving the page is

$$d \cdot l_2 + (d + k)l_0 - d \cdot l_1 \leq d \cdot l_0 + (d + k)l_0 \leq (2d + k)l_0.$$

In total, the expected amortized cost of the request is

$$l_0 + \frac{1}{k}(2d + k)l_0 \leq (\frac{2d}{k} + 2)l_0.$$

Next we analyze event II. Suppose that OPT moves the page from node $u_{OPT}$ to node $u'_{OPT}$. Let $l_0 = dist(u_{OPT}, u'_{OPT})$, $l_1 = dist(u_C, u_{OPT})$

and $l_2 = dist(u_C, u'_{OPT})$. OPT incurs a cost of $d \cdot l_0$. The change in potential is

$$(d + C)(l_2 - l_1) \leq (d + C)l_0.$$

The inequality follows again from the triangle inequality. The expected value of the global counter is $\frac{k+1}{2}$. Thus, the expected change in potential is

$$(d + \frac{k+1}{2})l_0.$$

This is $(1 + \frac{k+1}{2d})$ times the cost incurred by OPT. □

## 7.3 Page replication

It turns out that in terms of competitiveness, page replication is more complex than page migration. Online replication algorithms achieving a constant competitive ratio are only known for specific network topologies, such as trees, uniform networks and rings. A uniform network is a complete graph in which all edges have length 1. It was shown by Bartal *et al.* [16] that on general graphs, no deterministic or randomized online replication algorithm can a achieve a competitive ratio smaller than $\Omega(\log n)$, where $n$ is the number of nodes in the network.

On trees and uniform networks, the best deterministic online replication algorithm is 2-competitive, see Black and Sleator [21]. The best randomized algorithm achieves a competitive ratio of $\frac{e}{e-1} \approx 1.58$ against any oblivious adversary, see Albers and Koga [3]. Both upper bounds are tight. For rings, the best online replication algorithm currently known is 4-competitive [3].

In the following we concentrate on trees. The general online replication strategy given below was used in [21, 3]. We assume that initially only the root of the tree has a copy of the page.

**Algorithm COUNTER (for trees)**: The algorithm first chooses an integer $C$ from the range $[1, d]$. We will specify the exact value of $C$ later. While processing the request sequence, the algorithm maintains a counter for each node of the tree. Initially, all counters are set to 0. If there is a request at a node $v$ that does not have the page, then all counters along the path from $v$ to the closest node with the page are incremented by 1. When a counter reaches the value $C$, the page is replicated to the corresponding node.

In the above algorithm we can assume without loss of generality that whenever the page is replicated from node $u$ to node $v$, then the page is also replicated to all other nodes on the path from $u$ to $v$. This does not incur a higher cost.

**Theorem 24** *If $C = d$, then for any tree, the COUNTER algorithm is 2-competitive.*

**Theorem 25** *Suppose that $C$ is chosen randomly, where $C = i$, $1 \leq i \leq d$, with probability $p_i = \alpha \cdot \delta^{i-1}$. Here $\delta = (d+1)/d$ and $\alpha = (\delta - 1)/(\delta^d - 1)$. Then, for any tree, the COUNTER algorithm is $(\frac{\delta^d}{\delta^d - 1})$-competitive against any oblivious adversary.*

Note that $\frac{\delta^d}{\delta^d-1}$ goes to $\frac{e}{e-1} \approx 1.58$ as $d$ tends to infinity. We will only give a proof of Theorem 24.

**Proof of Theorem 24**: Let $r$ denote the root of the tree. We start with some observations regarding the COUNTER algorithm.

- At any time, the counters of the nodes on a path from $r$ to any other node in the tree are non-increasing.
- After each request, the nodes with the page are exactly those whose counters are (at least) $C$.
- If a node $v$ has the page, then all nodes on the path from $v$ to the root also have the page. Thus, the nodes with the page always form a connected component of the tree.

The above observations allow us to analyze the COUNTER algorithm as follows. We partition cost into parts corresponding to the edges of the tree. An edge $e$ incurs a cost (equal to the length of $e$) for a request if the path from the requested node to the closest node with the page passes through $e$. Otherwise the cost is 0. An edge also incurs the cost of a replication across it.

Consider an arbitrary edge $e$ and let $l(e)$ be the length of $e$. After $e$ has incurred $d$ times a cost of $l(e)$, a replication occurs across $e$ in the COUNTER algorithm.

Suppose that there are indeed $d$ requests on which $e$ incurs a cost. Then the total cost incurred by $e$ in the COUNTER algorithm is

$$2 \cdot d \cdot l(e).$$

There are at least $d$ requests at nodes below $e$ in the tree. Thus, with respect to OPT's cost, $e$ incurs a cost of $d \cdot l(e)$, and we obtain a cost ratio of 2.

Suppose that there are only $k$, $k < d$, requests on which $e$ incurs a cost in the COUNTER algorithm. The the total cost incurred by $e$ is

$$k \cdot l(e).$$

In OPT's cost, $e$ causes a cost of at least $k \cdot l(e)$ because there are at least $k$ requests at nodes below $e$. In this case we have a cost ratio of 1. □

Bartal *et al.* [16] and Awerbuch *et al.* [10] developed deterministic and randomized online algorithms for page replication on general graphs. The algorithms have an optimal competitive ratio of $O(\log n)$, where $n$ is the number of nodes in the graph. We describe the randomized algorithm [16].

**Algorithm COINFLIP**: If a requesting node $v$ does not have the page, then with probability $1/d$ the page is replicated to $v$ from the closest node with the page.

**Theorem 26** *The COINFLIP algorithm is $O(\log n)$-competitive on an arbitrary graph.*

## 7.4 Page allocation

In the last two sections we studied page migration and replication problems separately. It is also possible to investigate a combined version of migration and replication. Here we are allowed to maintain several copies of a page, even in the presence of write requests. However, if there is a write request, then all page replicas have to be updated at a cost. More precisely, the cost model is as follows. Consider an arbitrary graph.

- As usual, migrating or replicating a page from node $u$ to node $v$ incurs a cost of $dist(u, v)$.
- A page replica may be erased at 0 cost.
- If there is a read request at $v$ and $v$ does not have the page, then the incurred cost is $dist(u, v)$, where $u$ is the closest node with the page.
- The cost of a write request at node $v$ is equal to the cost of communicating from $v$ to all other nodes with a page replica.

This model was introduced and studied by Bartal *et al.* [16] and Awerbuch *et al.* [10] who presented deterministic and randomized online algorithms achieving an optimal competitive ratio of $O(\log n)$, where $n$ is the number of nodes in the graph. We describe the randomized solution [16].

**Algorithm COINFLIP**: If there is a read request at node $v$ and $v$ does not have the page, then with probability $\frac{1}{d}$, replicate the page to $v$. If there is a write request at node $v$, then with probability $\frac{1}{\sqrt{3d}}$, migrate the page to $v$ and erase all other page replicas.

**Theorem 27** *The COINFLIP algorithm is $O(\log n)$-competitive on an arbitrary graph with $n$ nodes.*

# 8 Scheduling and load balancing

## 8.1 Scheduling

The general situation in online scheduling is as follows. We are given a set of $m$ machines. A sequence of jobs $\sigma = J_1, J_2, \ldots, J_n$ arrives online. Each job $J_k$ has a processing $p_k$ time that may or may not be known in advance. As each job arrives, it has to be scheduled immediately on of the $m$ machines. The goal is to optimize, i.e. usually to minimize, a given objective function. There are many problem variants. We can study various machine types and various objective functions.

We consider a classical setting. Suppose that we are given $m$ *identical* machines. As each job arrives, its processing time is known in advance. The goal is to minimize the makespan, i.e., the completion time of the last job that finishes.

Graham [31] analyzed the GREEDY algorithm.

**Algorithm GREEDY**: Always assign a new job to the least loaded machine.

**Theorem 28** *The GREEDY algorithm is $(2 - \frac{1}{m})$-competitive.*

**Proof:** Given an arbitrary job sequence $\sigma$, let $T_G(\sigma)$ denote makespan of the schedule produced by GREEDY and let $T_{OPT}(\sigma)$ be the optimum makespan. Let $t_1$ denote the length of the time interval (starting from time 0) during which all machines are busy in the online schedule. Let $t_2 = T_G(\sigma) - t_1$. By the definition of the GREEDY algorithm, the job that finishes last in the GREEDY schedule starts at some time $t \le t_1$, i.e., its processing time is at least $t_2$. Thus, $t_2 \le \max_{1\le k\le n} p_k$. It is not hard to see that $t_1 \le \frac{1}{m}\sum_{k=1}^{n} p_k$. Clearly, $T_{OPT}(\sigma) \ge \max\{\frac{1}{m}\sum_{k=1}^{n} p_k, \max_{1\le k\le n} p_k\}$. Thus,

$$T_G(\sigma) = t_1 + t_2 \le 2\max\{\frac{1}{m}\sum_{k=1}^{n} p_k, \max_{1\le k\le n} p_k\} \le 2 \cdot T_{OPT}(\sigma)$$

and this shows that GREEDY is 2-competitive. A more refined analysis of $t_1$ and $t_2$ gives that competitive ratio is $2 - \frac{1}{m}$. Details are omitted here. □

Graham already analyzed GREEDY in 1966. It was unknown for a long time whether GREEDY achieves the best possible competitive ratio. Recently, a number of improved algorithms were presented. Bartal *et al.* [15] gave an algorithm that is 1.986-competitive. The best algorithm currently known is due to Karger *et al.* [38] and achieves a competitive ratio of 1.945. We describe their algorithm and first explain the intuition behind the improvement. The GREEDY algorithm always tries to maintain a flat schedule, i.e., all machine should be equally loaded. Problems arise if the schedule is completely flat and a large job comes in. On the other hand, the improved algorithms try to maintain some heavily loaded and some lightly loaded machines. Now, when a large job arrives, it can be assigned to a lightly loaded machine so that the makespan does not increase too much.

**Algorithm IMBALANCE**. Set $\alpha = 1.945$. A new incoming job $J_k$ is scheduled as follows. Let $h_i$ be the height of the $i$-th smallest machine, $1 \le i \le m$, and let $A_i$ be the average height of the $i-1$ smallest machines. Set $A_0 = \infty$. Schedule job $J_k$ on the tallest machine $j$ such that $h_j + p_k \le \alpha A_j$.

Next we discuss some extensions of the scheduling problem studied above.

**Identical machines, restricted assignment** We have a set of $m$ identical machines, but now each job can only be assigned to one of a subset of admissible machines. Azar *et al.* [9] showed that the GREEDY algorithm, which always assigns a new job to the least loaded machine among the admissible machines, is $O(\log m)$-competitive.

**Related machines** Each machine $i$ has a speed $s_i$, $1 \leq i \leq m$. The processing time of job $J_k$ on machine $i$ is equal to $p_k/s_i$. Aspnes *et al.* [6] showed that the GREEDY algorithm, that always assigns a new job to a machine so that the load after the assignment in minimized, is $O(\log m)$-competitive. They also presented an algorithm that is 8-competitive.

**Unrelated machines** The processing time of job $J_k$ on machine $i$ is $p_{k,i}$, $1 \leq k \leq n$, $1 \leq i \leq m$. Aspnes *et al.* [6] showed that GREEDY is only $m$-competitive. However, they also gave an algorithm that is $O(\log m)$-competitive.

## 8.2 Load balancing

In online load balancing we have again a set of $m$ machines and a sequence of jobs $\sigma = J_1, J_2, \ldots, J_n$ that arrives online. However, each job $J_k$ has a *weight* $w(k)$ and an unknown duration. For any time $t$, let $l_i(t)$ denote the load of machine $i$, $1 \leq i \leq m$, at time $t$, which is the sum of the weights of the jobs present on machine $i$ at time $t$. The goal is to minimize the maximum load that occurs during the processing of $\sigma$.

We concentrate on settings with $m$ identical machines. Azar and Epstein showed that the GREEDY algorithm is $(2 - \frac{1}{m})$-competitive. In the following we will study the situation with identical machines and restricted assignment, i.e., each job can only be assigned to a subset of admissible machines. Azar *et al.* [7] proved that GREEDY is $\Theta(m^{2/3})$-competitive. They also proved that no online algorithm can be better than $\Omega(\sqrt{m})$-competitive. Azar *et al.* [8] gave a matching upper bound. The algorithm is called ROBIN HOOD.

**Algorithm ROBIN HOOD**: Let $OPT$ be the optimum load achieved by the offline algorithm. ROBIN HOOD maintains an estimate $L$ for OPT satisfying $L \le OPT$. At any time $t$, machine $i$ is called *rich* if $l_i(t) \ge \sqrt{m}L$; otherwise machine $i$ is called *poor*. When a new job $J_k$ arrives, the estimate $L$ is updated, i.e.,

$$L := \max\{L, w(k), \frac{1}{m}(w(k) + \sum_{i=1}^{m} l_i(t))\}.$$

If possible, $J_k$ is assigned to a poor machine. Otherwise it is assigned to the rich machine that became rich most recently.

Azar *et al.* [8] analyzed the performance of ROBIN HOOD.

**Theorem 29** *ROBIN HOOD is $O(\sqrt{m})$-competitive.*

**Lemma 5** *At most $\lceil \sqrt{m} \rceil$ machine can be rich at any time.*

**Proof:** If more than $\lceil \sqrt{m} \rceil$ machines were rich, then the aggregate load on the machines would be greater than $\lceil \sqrt{m} \rceil \sqrt{m} L \ge mL$. However, by the definition of $L$, $L \le \frac{1}{m}(w(k) + \sum_{i=1}^{m} l_i(t))$, i.e., $mL$ is a lower bound on the aggregate load. □

**Lemma 6** *At all times $L \le OPT$.*

**Proof:** The proof is by induction on the number of assigned jobs. We only have to consider the times when $L$ changes. The lemma follows because $w(k) \le OPT$ and $\frac{1}{m}(w(k) + \sum_{i=1}^{m} l_i(t)) \le OPT$. □

**Proof of Theorem 29**: Consider a fixed time $t$. We show that for any machine $i$, $l_i(t) \le \lceil \sqrt{m} \rceil (L + OPT)$. If machine $i$ is poor, then the inequality is obvious. So suppose that machine $i$ is rich and let $t_0$ be the most recent time when machine $i$ became rich. Let $M(t_0)$ be the set of machines that are rich at time $t$ and the last instance at which they became rich is no later then $t_0$. Note that $i \in M(t_0)$.

Let $S$ be the set of jobs that were assigned to machine $i$ after $t_0$. All these jobs could only be scheduled on machines in $M(t_0)$. Let $j = |M(t_0)|$. We have $OPT \geq \frac{1}{j}\sum_{J_k \in S} w(k)$.

First suppose that $j \leq \lceil\sqrt{m}\rceil - 1$. Let $J_q$ be the job assigned to machine $i$ that caused machine $i$ to become rich. Then

$$l_i(t) \leq \lceil\sqrt{m}\rceil L + w(q) + \sum_{J_k \in S} w(k) \leq \lceil\sqrt{m}\rceil(L + OPT).$$

The last inequality follows because $w(q) \leq OPT$.

Now suppose that $j = \lceil\sqrt{m}\rceil$. Then $l_i(t_0) = \sqrt{m}L$ since otherwise the aggregate load in the system would exceed $mL$. Thus,

$$l_i(t) \leq \sqrt{m}L + \sum_{J_k \in S} w(k) \leq \lceil\sqrt{m}\rceil(L + OPT).$$

□

# 9 Robot navigation and exploration

Suppose that a robot is placed in an unknown environment. In *navigation problems* the robot has to find a path from a source point $s$ to a target $t$. In *exploration problems* the robot has to construct a complete map of the environment. In each case, the goal is to minimize the distance traveled by the robot.

In the following we concentrate on robot navigation and study a simple problem introduced by Baeza-Yates *et al.* [11]. Assume that the robot is placed on a line. It starts at some point $s$ and has to find a point $t$ on the line that is a distance of $n$ away. The robot is tactile, i.e., it has no vision and only knows that is has reached the target when it is actually located on $t$. We call a robot strategy $c$-competitive if the length of the path of the robot is at most $c$ times the distance between $s$ and $t$.

Since the robot does not know whether $t$ is located to the left or to the right of $s$, it should not move into one direction for a very long time. Rather, after the robot has traveled a certain distance into one direction, it should return to the start point $s$ and move into the other direction. For $i = 1, 2, \ldots$, let $f(i)$ be the distance walked by the robot before the $i$-th turn since its last visit to $s$. Baeza-Yates *et al.* [11] proposed a "doubling" strategy, i.e., they considered $f(i) = 2^i$. It is easy to verify that the total distance traveled by the robot is bounded by

$$2 \sum_{i=1}^{\lfloor \log n \rfloor + 1} 2^i + n \leq 9n.$$

Baeza-Yates *et al.* also proved that this robot strategy is optimal.

A more complex navigation problem can be described as follows. A robot is placed in a 2-dimensional scene with obstacles. It starts at some point $s$ and has to find a short path to a target $t$. When traveling through the scenes of obstacles, the robot always knows its current position and the position of $t$. However, the robot does not know the positions and extends of the obstacles in advance. Rather, it learns about the obstacles as it walks through the scene. Again we have tactile robot that learns about an obstacle by touching it. We call a robot strategy $c$-competitive, if for all scenes of obstacles, the length of the path traveled by the robot is at most $c$ times the shortest path from $s$ to $t$.

Most previous work on this problem has concentrated on the case that the obstacles are axis-parallel rectangles. Papadimitriou and Yannakakis [44] gave a lower bound.

**Theorem 30** *No deterministic online navigation algorithm in a general scene with $n$ rectangular, axis parallel obstacles can have a competitive ratio smaller than $\Omega(\sqrt{n})$.*

**Proof:** We consider a relaxed problem. A robot has to reach an arbitrary point on a vertical, infinitely long wall. The wall is a distance of $n$ away. We place long thin obstacles of width 1 and length $2n$ at integer coordinates. Whenever the robot circumvents an obstacle and makes a progress of 1 into $x$-direction, we place a new obstacle in front of him. After having placed $n$ obstacles, we stop this process. The robot has traveled a distance of at least $n \cdot \frac{n}{2}$.

Since $n$ obstacles were placed, the must be a $y$-coordinate of at most $n^{3/2}$ that has at most $\sqrt{n}$ obstacles. Thus, the optimal path to the wall has a length of $O(n^{3/2})$. □

Blum *et al.* [22] developed a deterministic online navigation algorithm that achieves a tight upper bound of $O(\sqrt{n})$, where $n$ is again the number of obstacles. Recently, Berman *et al.* [13] gave a randomized algorithm that is $O(n^{4/9} \log n)$-competitive against any oblivious adversary. An interesting open problem is to develop improved randomized online algorithms.

# References

[1] D. Achlioptas, M. Chrobak and J. Noga. Competitive analysis of randomized paging algorithms. To appear in the Fourth Annual European Symposium on Algorithms (ESA96), 1996.

[2] S. Albers. Improved randomized on-line algorithms for the list update problem. In *Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 412–419, 1995.

[3] S. Albers and H. Koga. New on-line algorithms for the page replication problem. In *Proc. of the 4th Scandinavian Workshop on Algorithm Theory*, Springer Lecture Notes in Computer Science, Volume 824, pages 25–36, 1994.

[4] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. In

*Proc. of the 23rd International Colloquium on Automata, Languages and Programming*, Springer Lecture Notes in Computer Science, Volume 1099, pages 514–525, 1996.

[5] S. Albers, B. von Stengel and R. Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters*, 56:135–139, 1995.

[6] J. Aspnes, Y. Azar A. Fiat, S. Plotkin and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proc. 25th ACM Annual ACM Symposium on the Theory of Computing*, pages 623–631, 1993.

[7] Y. Azar, A. Broder and A. Karlin. On-line load balancing. In *Proc. 36th IEEE Symposium on Foundations of Computer Science*, pages 218–225, 1992.

[8] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs and O. Waarts. Online load balancing of temporary tasks. In *Proc. Workshop on Algorithms and Data Structures*, Springer Lecture Notes in Computer Science, pages 119–130, 1993.

[9] Y. Azar, J. Naor and R. Rom. The competitiveness of on-line assignments. In *Proc. of the 3th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 203–210, 1992.

[10] B. Awerbuch, Y. Bartal and A. Fiat. Competitive distributed file allocation. In *Proc. 25th Annual ACM Symposium on Theory of Computing*, pages 164–173, 1993.

[11] R.A. Baeza-Yates, J.C. Culberson and G.J.E. Rawlins. Searching in the plane. *Information and Computation*, 106:234–252, 1993.

[12] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[13] P. Berman, A. Blum, A. Fiat, H. Karloff, A. Rosśen and M. Saks. Randomized robot navigation algorithm. In *Proc. of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 74–84, 1996.

[14] Y. Bartal, M. Charikar and P. Indyk. On page migration and other relaxed task systems. To appear in *Proc. of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[15] Y. Bartal, A. Fiat, H. Karloff and R. Vohra. New algorithms for an ancient scheduling problem. In *Proc. 24th Annual ACM Symposium on Theory of Computing*, pages 51–58, 1992.

[16] Y. Bartal, A. Fiat and Y. Rabani. Competitive algorithms for distributed data management. In *Proc. 24th Annual ACM Symposium on Theory of Computing*, pages 39–50, 1992.

[17] S. Ben-David, A. Borodin, R.M. Karp, G. Tardos and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:2-14,1994.

[18] J.L. Bentley, K.L. Clarkson and D.B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 179–187, 1990.

[19] J.L. Bentley and C.C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communication of the ACM*, 28:404–411, 1985.

[20] J.L. Bentley, D.S. Sleator, R.E. Tarjan and V.K. Wei. A locally adaptive data compression scheme. *Communication of the ACM*, 29:320–330, 1986.

[21] D.L. Black and D.D. Sleator. Competitive algorithms for replication and migration problems. Technical Report Carnegie Mellon University, CMU-CS-89-201, 1989.

[22] A. Blum, P. Raghavan and B. Schieber. Navigating in unfamiliar geometric terrain. In *Proc. 23th Annual ACM Symposium on Theory of Computing*, pages 494–504, 1991.

[23] A. Borodin, S. Irani, P. Raghavan and B. Schieber. Competitive paging with locality of reference. In *Proc. of the 23rd Annual ACM Symposium on Theory of Computing*, pages 249–259, 1991.

[24] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. DEC SRC Research Report 124, 1994.

[25] F.R.K. Chung, D.J. Hajela and P.D. Seymour. Self-organzing sequential search and Hilbert's inequality. *Proc. 17th Annual Symposium on the Theory of Computing*, pages 217–223, 1985..

[26] P. Elias. Universal codeword sets and the representation of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

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

[28] G.H. Gonnet, J.I. Munro and H. Suwanda. Towards self-organizing linear search. In *Proc. 19th Annual IEEE Symposium on Foundations of Computer Science*, pages, 169–174, 1979.

[29] R.G. Gallager. *Information Theory and Reliable Communication*. Wiley, New York, 1968.

[30] M.J. Golin. PhD thesis, Department of Computer Science, Princeton University, 1990. Technical Report CS-TR-266-90.

[31] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[32] E.F. Grove. The Harmonic online $k$-server algorithm is competitive. In *Proc. of the 23rd Annual ACM Symposium on Theory of Computing*, pages 260–266, 1991.

[33] G.H. Hardy, J.E. Littlewood and G. Polya. *Inequalities.* Cambridge University Press. Cambridge, England, 1967.

[34] S. Irani. Two results on the list update problem. *Information Processing Letters*, 38:301–306, 1991.

[35] S. Irani. Corrected version of the SPLIT algorithm. Manuscript, January 1996.

[36] S. Irani, A.R. Karlin and S. Phillips. Strongly competitive algorithms for paging with locality of reference. In *Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 228–236, 1992.

[37] E. Koutsoupias and C.H. Papadimitriou. On the $k$-server conjecture. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 507–511, 1994.

[38] D. Karger, S. Phillips and E. Torng. A better algorithm for an ancient scheduling problem. In *Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 132–140, 1994.

[39] R. Karp and P. Raghavan. From a personal communication cited in [46].

[40] C. Lund, N. Reingold, J. Westbrook and D. Yan. On-line distributed data management. In *Proc. of the 2nd Annual European Symposium on Algorithms*, Springer LNCS Vol. 855, pages 202–214, 1994.

[41] M.S. Manasse, L.A. McGeoch and D.D. Sleator. Competitive algorithms for on-line problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 322-33, 1988.

[42] L.A. McGeoch and D.D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

[43] R. Motwani and P. Raghavan. Randomized Algorithms, Cambridge University Press, 1995.

[44] C.H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.

[45] P. Raghavan and M. Snir. Memory versus randomization in online algorithms. In *Proc. 16th International Colloquium on Automata, Languages and Programming*, Springer Lecture Notes in Computer Science, Vol. 372, pages 687–703,1989.

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

[47] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19:63–67, 1997.

[48] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communication of the ACM*, 28:202–208, 1985.

[49] B. Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47:5–9, 1993.

[50] R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[51] J. Westbrook. Randomized algorithms for the multiprocessor page migration. *SIAM Journal on Computing*, **23**:951–965, 1994.

[52] I.H. Witten and T. Bell. The Calgary/Canterbury text compression corpus. Anonymous ftp from ftp.cpsc.ucalgary.ca : /pub/text.compression/corpus/ text.compression.corpus.tar.Z.

[53] A.C.-C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. 17th Annual IEEE Symposium on Foundations of Computer Science*, pages 222–227, 1977.

## Recent Publications in the BRICS Lecture Series

**LS-96-2 Susanne Albers. *Competitive Online Algorithms*. September 1996. iix+57 pp.**

**LS-96-1 Lars Arge. *External-Memory Algorithms with Applications in Geographic Information Systems*. September 1996. iix+52 pp.**

**LS-95-5 Devdatt P. Dubhashi. *Complexity of Logical Theories*. September 1995. x+46 pp.**

**LS-95-4 Dany Breslauer and Devdatt P. Dubhashi. *Combinatorics for Computer Scientists*. August 1995. viii+184 pp.**

**LS-95-3 Michael I. Schwartzbach. *Polymorphic Type Inference*. June 1995. viii+24 pp.**

**LS-95-2 Sven Skyum. *Introduction to Parallel Algorithms*. June 1995. viii+17 pp. Second Edition.**

**LS-95-1 Jaap van Oosten. *Basic Category Theory*. January 1995. vi+75 pp.**