The On-Line K-Server Problem

Aris Floratos and Ravi Boppana
Courant Institute of Mathematical Sciences, NYU

Abstract

We survey the research performed during the last few years on the on-line k-server problem over metric spaces. A variety of algorithms are presented — both deterministic and randomized — and their performance is studied in the framework of competitive analysis. Restrictions of the problem to special cases of metric spaces are also considered.

1 Introduction

In much of the theory of algorithm analysis the following fundamental assumption is made: whenever an algorithm is called upon to solve any particular instance of a problem, all the input necessary for the solution is available at the time the algorithm begins its computation. There are situations, though, where this assumption is just not realistic. In many cases the nature of a problem dictates that the input has to be presented to the algorithm incrementally, one piece at a time, and the algorithm must produce a corresponding piece of output based only on what has been seen up to that point. The optimal response though, might depend on the way that future input (unknown at the time when a decision must be made) is structured.

A typical example of such a situation (drawn from the field of computer architecture) is the caching problem. In this setting, we have two different kinds of memories: one (called the "cache") is small and fast while the other (the "main memory") is large and slow. Each memory can store a number of pages. At any point, the cache contains a subset of the pages stored in main memory. The input consists of a sequence of page references. Whenever such a reference arrives, the cache is checked for the corresponding page. If the page is found there, then a *hit* is said to have occured and the reference can be served at no cost. If, however, the page is not in the cache, then we have a *miss*. In this case, one of the pages already in the cache must be selected and *evicted*, i.e. replaced by the just referenced page. This replacement process entails a certain cost, since data must be transfered between the main memory and the cache. What is needed in the caching problem is a *page replacement algorithm*, an algorithm that decides in case of a miss which page to evict from the cache. The objective of such an algorithm is to minimize the overall cost of processing any given page reference sequence (or, equivalently, to minimize the total number of misses). As is very well known, this cost minimization is achieved if, whenever a miss occurs, the cache page evicted is the one that will not be used for the longest time in the future. This replacement strategy is, however, inherently *off-line*: deciding which page to evict requires knowledge of future page references. In practice, knowledge of the future is a luxury we cannot afford. Any realistic page replacement algorithm must make its decisions *on-line*: as soon as a miss occurs, a page replacement must take place before any further references arrive.

The caching problem described above is just one instance of a big collection of problems coming from a great variety of disciplines such as computer science ([24]), mathematics ([10]), economics ([22]) and many others. All these problems, despite their great diversity, share a number of common features:

- They all are *optimization* problems. That is, given an instance (the input) of any of these problems, what is asked for is an output of minimum cost (or maximum return, depending on the problem).

- Although it is, in general, easy to construct *off-line algorithms* for these problems (i.e. algorithms which, given a certain input, will compute an/the optimal output), what we are interested in is the design of *on-line algorithms*. Algorithms which will be fed their input in a piece-meal manner and that must produce an output in a similarly incremental way: as soon as some chunk of input becomes available, a piece of output must be produced.

Such problems, calling for the design of on-line algorithms, are categorized under the broad class of *on-line problems*. Their study over the last few years has enjoyed an increased attention, especially after the seminal work of Sleator and Tarjan in [24].

The questions that are usually of interest when faced with an on-line problem are the following:

> What is a good on-line algorithm for the problem? (design)
>
> How well does a specific algorithm perform, i.e. how close to optimal will the solutions produced by the algorithm be? (performance analysis)
>
> What is the best possible performance that an on-line algorithm for the problem can achieve? (lower bounds)
>
> Is it possible to get better results by using randomized algorithms? (randomization vs. determinism)

In this survey these questions are addressed from the perspective of a particular on-line problem, the *k-server problem*. Our intention, apart from presenting the results known to date, is also to exhibit some of the techniques used in the research of on-line problems.

More specifically, the structure of the paper is as follows. In Section 2 the k-server problem is defined and competitive analysis, the framework where the problem is studied, is introduced. In Section 3 the problem is considered in its most general setting, making no assumptions about the structure of the underlying metric space. Sections 4 to 7 describe several algorithms for restricted metric spaces, exhibiting how such restrictions can be exploited to obtain optimal on-line k-server algorithms. In Section 8 we present some lower bounds showing what kind of performance cannot be achieved by on-line algorithms. Finally, Section 9 contains a discussion about questions that still remain unanswered.

2 The k-server problem

The on-line k-server problem, introduced by Manasse et al. [18], has virtually defined the area of on-line problems. It is set on a *metric space* inhabited by k servers. Initially, each server is positioned at some point of the space. Over time, requests arrive for service at points of the space. Immediately after a request at some point q of the space comes in, a server must be moved to q (if none is there) in order to serve the request. Besides the server that moves to q, the other servers are also free to move (e.g. to position themselves favorably for handling subsequent requests). When a server moves it incurs a cost equal to the distance it covers. The cost of a request is equal to the sum of the costs incured by all servers during the service of that request. The cost of a sequence of requests is equal to the sum of the costs of all requests in the sequence. Our goal under this setting is to design on-line algorithms which will decide which server(s) to move when a request arrives so that any sequence of requests can be served with a cost as small as possible.

We now introduce some definitions and terminology to help formalize the above discussion.

A metric space is a pair (Q, d), where Q is a set of points and $d : Q \times Q \to \Re^+$ is a non-negative distance function satisfying the *triangle inequality*:

$$d(x, y) + d(y, z) \geq d(x, z) \qquad \text{for all x, y, z in } Q.$$

If $d(x, y) = d(y, x)$ for all x, y in Q then the space is called *symmetric*. From now on and unless otherwise specified the term "metric space" will be used to denote a symmetric space.

A *k-configuration* for a metric space (Q, d) is any multiset C of size k over Q. Given two k-configurations C and C' of a metric space (Q, d) we define a matching between C and C' to be a bijection between C and C'. The *matching distance* of such a matching is the sum of the distances of all matched pairs of points. The notation $|CC'|$ indicates the minimum matching distance achievable for the configurations C and C'.

Finally a request sequence $\boldsymbol{\rho} = (r_1, r_2, \ldots, r_m)$, of size m, is any element of Q^m. Such a sequence defines a set of requests and the order of their arrival.

An instance of the k-server problem is defined by specifying the metric space (Q, d) and the number k of the servers. Let A be an on-line algorithm for that instance. The input to the algorithm is a k-configuration C_0 and a request sequence $\boldsymbol{\rho} = (r_1, r_2, \ldots, r_m)$. Algorithm A is started with the k servers positioned at the points prescribed by C_0 (which is called the *initial configuration*). Then the requests of $\boldsymbol{\rho}$ start arriving, one after the other. When r_i comes in, the position of the servers is described by some configuration C_{i-1}. In order to serve r_i, algorithm A must produce a new k-configuration C_i containing the point r_i and move the servers to this new configuration. The output of A on $\boldsymbol{\rho}$ is the sequence C_1, C_2, $\ldots$, C_m of k-configurations it generates in serving the requests of $\boldsymbol{\rho}$. The cost $\mathrm{cost}_A(r_i)$ of serving r_i under A is defined as

$$\mathrm{cost}_A(r_i) = |C_{i-1} C_i|.$$

The cost of serving the overall sequence $\boldsymbol{\rho}$ under A is accordingly defined as:

$$\mathrm{cost}_A(\boldsymbol{\rho}) = \sum_{i=1}^{m} \mathrm{cost}_A(r_i).$$

The above discussion can be extended to also include randomized on-line algorithms. When such an algorithm is presented with a request r_i it can choose its response C_i from a probability distribution of k-configurations, all including the point r_i. As a consequence, each of the configurations C_1, C_2, $\ldots$, C_m that constitute the output of the algorithm to a request sequence $\boldsymbol{\rho}$ is really a random variable. When studying randomized on-line algorithms for the k-server problem the quantity we are interested in is the *expected cost* of serving a request sequence.

We now turn to the question of how to analyze the performance of an on-line algorithm for the k-server problem.

2.1 Competitive analysis

As is probably clear by now, the main impact of the requirement to process the input as it arrives is that for any on-line algorithm there will be input sequences forcing the algorithm to produce sub-optimal answers. A question then arises as to what would be an appropriate performance measure for studying and comparing the power of such algorithms. Sleator and Tarjan in [24] answered that question by suggesting that the solution produced by an on-line algorithm on some input should be compared to the optimal solution for that input. Manasse et al. [18] formalized this idea and introduced the notion of *competitiveness*. Here this notion is presented in the context of the on-line k-server problem but it applies to any on-line problem (with the appropriate problem-specific adaptations).

Definition 2.1 *An on-line algorithm A for the k-server problem is called c-competitive if for any initial configuration C_0 and any input sequence $\boldsymbol{\rho}$:*

$$cost_A(\boldsymbol{\rho}) \leq c \cdot cost_{OPT}(\boldsymbol{\rho}) + I(C_0)$$

where $c > 0$, $cost_{OPT}(\boldsymbol{\rho})$ is the cost that the optimal off-line algorithm pays for processing $\boldsymbol{\rho}$ starting at C_0 and I is a non-negative function depending only on the initial configuration.

According to the above definition, showing that an on-line algorithm A is c-competitive will guarantee that the cost that A pays in processing any sequence of requests $\boldsymbol{\rho}$ is never greater than c times what an optimal off-line algorithm would pay for the same sequence plus an additive term dependent only on the starting configuration and not on the sequence or its length.

In fact, the definition of competitive ratio in [18] is slightly different than the one given here. In that work an on-line algorithm A is called c-competitive if the inequality of Definition 2.1 holds for every off-line algorithm. That is, A is called c-competitive if for every off-line algorithm B:

$$\mathrm{cost}_A(\boldsymbol{\rho}) \leq c \cdot \mathrm{cost}_B(\boldsymbol{\rho}) + I_B(C_0)$$

Although this definition is really no different than the one given here, it is worth mentioning because it is closely connected to the method used in proving competitiveness results. The usual approach is to consider the k-server problem as a game between the on-line algorithm A under consideration and an *adversary* B. The advesary has its own set of k servers and is supposed to know the code of algorithm A. The role of B is then to generate and service a sequence of requests $\boldsymbol{\rho}$ so as to maximize the ratio $\mathrm{cost}_A(\boldsymbol{\rho})/\mathrm{cost}_B(\boldsymbol{\rho})$. To prove that A is c-competitive it is enough to prove that for any adversary B and for any sequence $\boldsymbol{\rho}$ generated by B the inequality above holds.

This view of the problem as a game between the on-line algorithm and an adversary is also used for defining the competitive ratio of a randomized on-line algorithm. However, things are a little more tricky in this case. When the on-line algorithm A is deterministic, the adversary B can produce a request sequence without having to wait for A to respond to a request before generating the next one: since A is deterministic and B knows the code of A, the adversary can predict the response of A over the complete sequence. This is not, though, the case with a randomized algorithm A. Even by knowing the code of A, the adversary B cannot predict what the exact response of A to a given request will be. This observation has motivated the definition of adversaries with varying power. There are three kinds of adversaries that have been proposed for analyzing randomized on-line algorithms:

Oblivious: An adversary of this type will in advance generate and serve the complete sequence of requests before the on-line algorithm makes any move.

Adaptive on-line: Such an adversary will generate the next request based on how the algorithm answered the previous request. It will also serve the new request immediately (i.e. without waiting to see how the algorithm will respond to the request).

Adaptive off-line: An adaptive off-line adversary will also generate the next request based on how the algorithm answered the previous one. It will, however, serve the whole request sequence optimally, at the end.

Definition 2.2 *A randomized on-line algorithm A for the k-server problem is called c-competitive against any oblivious (adaptive on-line, adaptive off-line) adversary if for every oblivious (adaptive on-line, adaptive-off line) adversary B, every initial configuration C_0 and every request sequence $\boldsymbol{\rho}$:*

$$E[cost_A(\boldsymbol{\rho})] \leq c \cdot E[cost_B(\boldsymbol{\rho})] + I_B(C_0).$$

From the description of the adversaries it can be seen that every oblivious adversary can be simulated by an adaptive on-line adversary and that every adaptive on-line adversary can, in turn, be simulated by some adaptive off-line adversary. If c_O, c_N, c_F are the competitive ratios of an on-line algorithm A against any oblivious, adaptive on-line and adaptive off-line adversary respectively, it then follows that $c_O \leq c_N \leq c_F$. The reverse order of simulations also hold when A is a deterministic algorithm (in which case $c_O = c_N = c_F$). This is not true, though, when playing against randomized algorithms. The power that is, in a sense, taken away from an oblivious adversary by allowing the on-line algorithm to use randomization, is returned back when permitting the adversary to watch the replies of the algorithm before generating the next request. As a result, there exist randomized on-line algorithms for which $c_O < c_N < c_F$. In quantifying the amount of extra power allowed by each kind of adversary, Ben-David et al. ([1]) proved the following.

Lemma 2.1 *If the randomized algorithm A is c-competitive against any adaptive on-line adversary and there exists a randomized algorithm that is c'-competitive against any oblivious adversary, then A is $c \cdot c'$-competitive against any adaptive off-line adversary.*

Lemma 2.2 *If the randomized algorithm A is c-competitive against any adaptive off-line adversary then there exists a deterministic c-competitive algorithm.*

The last lemma shows that randomization adds no extra power against adaptive off-line adversaries. Furthermore, combining both these lemmas, it can be infered that the existence of a randomized on-line algorithm which is c-competitive against any adaptive on-line adversary implies that there also exists a c^2-competititve deterministic algorithm.

All the algorithms presented in this survey are studied in the context of the competitive analysis outlined above. Before proceeding with the actual presentation, though, there is one final remark that we would like to make. Part of the reason why the on-line k-server problem has drawn a great attention in recent years is the fact that it closely models several important real-life problems. Caching, mentioned in the introduction, is one of them. The servers in this case are the cache page frames, the points of the metric space are the various pages and the distance between each pair of points is one[1]. Every page reference then is a request to a point of the space which is either covered (a hit) or uncovered (a miss) by a server. Serving input/output requests on a disk with multiple read/write heads is another example. Due to this practical dimension of the problem one might be interested in other properties of an on-line algorithm such as ease of implementation or speed. Such complexity questions, though, fall outside the context of competitive analysis. The application-oriented reader might want to keep this fact in mind when evaluating the algorithms presented here.

3 Unrestricted metric spaces

In analyzing on-line algorithms for the k-server problem, any special properties of the underlying metric space may prove crucial in the effort to obtain a good competitive ratio for the algorithm under consideration. Examples of this fact will be exhibited in subsequent sections. For now, though, we assume no extra properties other than those described in the definition of metric spaces. In this context two on-line algorithms are presented, one deterministic and one randomized.

The deterministic, called the *work function algorithm*, is shown to be $(2k-1)$-competitive. This is the best competitive ratio that has been shown for any deterministic algorithm[2] in unrestricted metric spaces and is very close to a lower bound proved by Mannasse et al. in [18]. It was shown therein that no *deterministic* on-line k-server algorithm can be better than k-competitive. In the same work, k is conjectured to be a tight lower bound. This is the famous *k-server conjecture*. It states that for every symmetric space there exists a deterministic on-line algorithm for the k-server problem which is k-competitive. Proving (or disproving) that conjecture remains the foremost open question in the area.

The randomized algorithm presented here, called the *HARMONIC algorithm*, is shown to have a competitive ratio that is exponential in k against an adaptive on-line adversary. This is the only randomized algorithm for unrestricted metric spaces that has appeared in the literature.

[1]The value 'one' is just a normalization constant for the data transfer cost.

[2]Before the proof for the work function algorithm the best competitive ratio known ([12]) for deterministic algorithms in unrestricted spaces was exponential in k.

3.1 Work-function algorithm

The work-function algorithm was originally proposed by Chrobak and Larmore in [6]. It was shown to be $(2k-1)$-competitive by Koutsoupias and Papadimitriou in [16]. In this subsection we present an outline of the proof given in the later paper.

The definition of the algorithm is based on the concept of the *work function*. Let C_0 indicate an initial configuration for the k servers and consider any request sequence $\boldsymbol{\rho} = (r_1, r_2, \ldots, r_m)$.

Definition 3.1 (Work function) *For any k-configuration C let $w_i(C)$ denote the optimal cost of serving the subsequence of requests $\boldsymbol{\rho^i} = (r_1, r_2, \ldots, r_i)$ starting at C_0 under the requirement that at the end the servers must be placed at the configuration C. The non-negative real function w_i defined on the set of k-configurations is called the work function after serving $\boldsymbol{\rho^i}$.*

The following properties of work functions can be directly deduced from the above definition and the metric property of the underlying space:

Corollary 3.1 *1. For every configuration C,*

$$w_i(C) = \begin{cases} w_{i-1}(C) & \text{if } r_i \in C \\ \min_{s \in C}\{w_i(C - s + r_i) + d(s, r_i)\} & \text{otherwise} \end{cases}$$

where $(C - s + r_i)$ indicates the k-configuration resulting from C by replacing the point s in C by r_i.

2. For every configuration C,

$$w_i(C) \geq w_{i-1}(C)$$

3. For every pair of configurations C and C',

$$w_i(C) \leq w_i(C') + |CC'|$$

4. For every configuration C,

$$w_0(C) = |C_0 C|$$

5. For every configuration C and any sequence of requests $\boldsymbol{\rho} = (r_1, r_2, \ldots, r_m)$,

$$cost_{OPT}(\boldsymbol{\rho}) \leq \ w_m(C)$$

The work function algorithm (WFA) is defined as follows:

Definition 3.2 (Work function algorithm) *Let C_{i-1} be the configuration of WFA's servers at the time when r_i arrives. The server*[3] *$s_i \in C_{i-1}$ that will move to service the new request is the one minimizing the quantity*

$$w_i(C_{i-1} - s_i + r_i) + d(s_i, r_i).$$

Thus, $C_i = C_{i-1} - s_i + r_i$, for the s_i specified above.

The cost $\mathrm{cost}_{WFA}(\boldsymbol{\rho})$ that WFA pays for serving a sequence of requests $\boldsymbol{\rho} = (r_1, r_2, \ldots, r_m)$ is

$$\mathrm{cost}_{WFA}(\boldsymbol{\rho}) = \sum_{i=1}^{m} |C_{i-1}C_i| = \sum_{i=1}^{m} d(s_i, r_i)$$

where the C_i and s_i are computed as described in the definition of the work function algorithm. It turns out that instead of directly studying this cost it is more convenient to look at an "inflated" version of it, called the *extended cost*:

Definition 3.3 (Extended cost) *The extended cost of serving the i-th request in the sequence $\boldsymbol{\rho}$ is*

$$\psi_i = \max_C \{w_i(C) - w_{i-1}(C)\}$$

while the total extended cost of serving the entire sequence is

$$\Psi(\boldsymbol{\rho}) = \sum_{i=1}^{m} \psi_i.$$

The use of the extended cost is illuminated by the following lemma:

Lemma 3.1 *If there exists a function I such that for all request sequences $\boldsymbol{\rho}$ and all starting configurations C_0*

$$\Psi(\boldsymbol{\rho}) \le \ (c+1)\ cost_{OPT}(\boldsymbol{\rho}) + I(C_0)$$

then the work function algorithm is c-competitive.

Proof: It is enough to show that $\Psi(\boldsymbol{\rho}) \ge \ \mathrm{cost}_{WFA}(\boldsymbol{\rho}) + \mathrm{cost}_{OPT}(\boldsymbol{\rho})$.

$$\begin{aligned}
\Psi(\boldsymbol{\rho}) &= \sum_{i=1}^{m} \max_C \{w_i(C) - w_{i-1}(C)\} \\
&\ge \sum_{i=1}^{m} (w_i(C_{i-1}) - w_{i-1}(C_{i-1})) \\
&= \sum_{i=1}^{m} (w_i(C_i) + d(s_i, r_i) - w_{i-1}(C_{i-1}))
\end{aligned}$$

[3] For brevity, a server will be sometimes identified with the point it occupies.

$$\begin{aligned} &= \sum_{i=1}^{m} d(s_i, r_i) \;+\; \sum_{i=1}^{m} (w_i(C_i) - w_{i-1}(C_{i-1})) \\ &= \text{cost}_{WFA}(\boldsymbol{\rho}) + w_m(C_m) - w_0(C_0) \\ &\geq \text{cost}_{WFA} + \text{cost}_{OPT}(\boldsymbol{\rho}) \end{aligned}$$

where the C_i and s_i are the ones computed by the work function algorithm. □

The remaining of the subsection is devoted in proving that the inequality of Lemma 3.1 holds for $c = 2k - 1$. The proof is carried through by the use of an appropriately defined potential function. Before exhibiting this potential function though, the following definition and lemma are needed.

Definition 3.4 *For any work function w_i and any point q of the metric space, the configuration C that minimizes the expression*

$$w_i(C) - \sum_{x \in C} d(x, q)$$

is called the minimizer of q with respect to w_i.

Lemma 3.2 *Consider the request r_i of the input request sequence $\boldsymbol{\rho}$ and let C be the minimizer of r_i with respect to the work function w_i. For any configuration C'*

$$w_i(C) - w_{i-1}(C) \geq \; w_i(C') - w_{i-1}(C').$$

That is, the extended cost ψ_i of serving r_i is achieved at C.

The proof of the above lemma (which is not presented here) is based on a property shared by all work functions, the *quasiconvexity property.* This property states that for every work function w_i and every pair of configurations C and C' there exists a bijection $h : C \to C'$ such that for every partition of C into X, Y:

$$w_i(C) + w_i(C') \geq \; w_i(X \cup h(Y)) + w_i(Y \cup h(X)).$$

With all the above in place, we are now ready to conclude the proof of the claimed competitive ratio for the work function algorithm.

For any configurations $U = (u_1, u_2, \ldots, u_k)$ and $B_j = (b_{j1}, b_{j2}, \ldots, b_{jk})$, $j = 1, \ldots, k$, and any work function w_i, let

$$\Gamma(w_i, U, B_1, \ldots, B_k) = kw_i(U) + \sum_{j=1}^{k} (w_i(B_j) - \sum_{l=1}^{k} d(u_j, b_{jl}))$$

Define $\Phi(w_i)$ to be the minimum value of $\Gamma(w_i, U, B_1, \ldots, B_k)$ over all configurations $U, B_1, \ldots, B_k$.

Lemma 3.3 *For every work function w_i the minimum value of $\Gamma(w_i, U, B_1, \ldots, B_k)$ is achieved for some U containing r_i.*

Proof: Fix some $U, B_1, \ldots, B_k$ for which the minimum value of $\Gamma(w_i, U, B_1, \ldots, B_k)$ is achieved. If $r_i \notin U$ then by the definition of work function there exists some u_h in U such that

$$w_i(U) = w_i(U - u_h + r_i) + d(u_h, r_i).$$

It can then be seen, using the triangle inequality, that by replacing U by $U - u_h + r_i$ in the parameters of Γ the value of the function does not increase. $\Box$

Lemma 3.4 *The work function algorithm is* $(2k-1)$*-competitive.*

Proof: Let A be the minimizer of request r_i with respect to w_i. Let $\Phi(w_i) = \Gamma(w_i, U, B_1, \ldots, B_k)$ for some $U, B_1, \ldots, B_k$. By to Lemma 3.3 it can be assumed that $r_i = u_h \in U$, for some h. Since A is the minimizer of r_i with respect to w_i we can also assume that $A = B_h$. Let Γ_{w_i}, $\Gamma_{w_{i-1}}$ be the values of function Γ computed at w_i, w_{i-1} for the configurations $U, B_1, \ldots, B_k$ described above. Using property 2 of Corollary 3.1:

$$\Phi(w_i) - \Phi(w_{i-1}) \geq \Gamma_{w_i} - \Gamma_{w_{i-1}} \geq w_i(A) - w_{i-1}(A) = \psi_i.$$

Summing over all i:

$$\Phi(w_m) - \Phi(w_0) \geq \Psi(\boldsymbol{\rho}).$$

Let C_{OPT} be the configuration in which the optimal off-line algorithm ends up after serving the input request sequence $\boldsymbol{\rho}$. From the definition of the potential function Φ it can then be seen that

$$\Phi(w_m) \leq \Gamma(w_m, C_{OPT}, C_{OPT}, \ldots, C_{OPT}) \leq 2k\, w_m(C_{OPT}) = 2k \operatorname{cost}_{OPT}(\boldsymbol{\rho}).$$

By also observing that $\Phi(w_0) = -\sum_{x,y \in C_0} d(x,y) = -I(C_0)$ we get

$$\Psi(\boldsymbol{\rho}) \leq 2k \operatorname{cost}_{OPT}(\boldsymbol{\rho}) + I(C_0)$$

which, by Lemma 3.1, concludes the proof. $\Box$

3.2 The HARMONIC algorithm

The HARMONIC algorithm, originally proposed by Raghavan and Snir in [23], is a simple and natural randomized algorithm. It dictates that a server will be selected to serve a new request with a probability inversely proportional to its distance from the request. More formally:

Definition 3.5 (HARMONIC) *Let $C = \{h_1, h_2, \ldots, h_k\}$ be the k-configuration describing the positions of the servers at the time when a new request r arrives.*

If $r \in C$ then the requested point is already covered. Otherwise the server h_i will be selected to serve r with probability $\frac{1/d(h_i,r)}{N}$, where

$$N = \sum_{i=1}^{k} \frac{1}{d(h_i, r)}.$$

One of the appealing properties of the HARMONIC algorithm is the fact that it is *memoryless*: the probability distribution of the possible responses to an incoming request depends only on the configuration of the servers and the current request and not on past events. As a consequence, it takes only $O(k)$ time to decide which server to move (this is to be contrasted with the work function algorithm which requires $\Omega(k\binom{i+k-1}{k-1})$ time for computing its response to the i-th request). It is this property that makes HARMONIC a good candidate for implementation and is also part of the reason why its analysis has drawn quite some attention. In [23] where it was introduced it is shown to be $k(k+1)$-competitive against any adaptive on-line adversary for the special case when the underlying metric space has $(k+1)$ points. The first attempt to analyze the performance of the HARMONIC algorithm for spaces with arbitrary size and for any value of k was made by Berman et al. in [2]. It fell short on the "any value of k" part. Specifically, they were able to prove that HARMONIC is competitive against any on-line adversary in every symmetric space but only for $k = 3$. The competitive factor turned out to be 3^{17000}(!). Their approach was subsequently used by Chrobak and Larmore ([17]) to prove HARMONIC 3-competitive against any adaptive on-line adversary for the case where $k = 2$. Grove in [13] was the first one to succesfully show that HARMONIC is competitive in every symmetric space and for all values of k. The competitive ratio achieved was $\frac{5}{4}k2^k - 2k$. In what follows we present an outline of the method used in that work.

The proof proceeds by assuming that HARMONIC is "playing" against an adaptive on-line adversary which generates the request sequence $\boldsymbol{\rho} = (r_1, r_2, \ldots, r_m)$. The k-configuration C_i (A_i) will be used to denote the position of the HARMONIC (adversary) servers *after* HARMONIC (the adversary) has served request r_i. Initially, the servers of both the adversary and the HARMONIC start at the same configuration, so $C_0 = A_0$. The game consists of m phases. At the begining of phase i the adversary will choose a new request point r_i and move one of its servers to that point thus changing its configuration from A_{i-1} to A_i and paying a cost equal to $\text{cost}_{ADV}(r_i) = |A_{i-1}A_i|$. The phase is concluded by the HARMONIC algorithm which has to choose one of its servers and move it to r_i, paying a cost $\text{cost}_{HAR}(r_i) = |C_{i-1}C_i|$, where C_{i-1}, C_i are analogous to A_{i-1} and A_i. What must be realized here is that, since the adversary chooses a request point based on the responses of the HARMONIC algorithm up to that point and due to the randomized nature of HARMONIC, all the quantities involved (the requests r_i, the configurations C_i and A_i, the costs $\text{cost}_{HAR}(r_i)$

and $\text{cost}_{ADV}(r_i)$) are random variables.

The proof proceeds by defining an apropriate potential function Φ describing the state of the system as it moves from phase to phase. The arguments of Φ are the current configurations of the HARMONIC algorithm and the adversary. The aim is to show that the potential function has the following properties:

Property 1: It is non-negative.

Property 2: When the adversary moves a server at the begining of phase i, the potential function does not increase by more than c times (for some c to be specified) the cost paid by the adversary during that move:

$$\Phi(C_{i-1}, A_i) - \Phi(C_{i-1}, A_{i-1}) \leq c \cdot cost_{ADV}(r_i).$$

Property 3: When HARMONIC moves a server to service request r_i, the expected drop in the potential is large enough to cover the expected cost paid by HARMONIC in serving r_i:

$$E[cost_{HAR}(r_i)] + E[\Phi(C_i, A_i) - \Phi(C_{i-1}, A_i)] \leq 0.$$

Lemma 3.5 *If Φ has the above properties, then HARMONIC is c-competitive against any adaptive on-line aversary.*

Proof: Using the properties above it can be seen that:

$$E[cost_{HAR}(\boldsymbol{\rho})] = \sum_{i=1}^{m} E[cost_{HAR}(r_i)] \leq c\, E[cost_{ADV}(\boldsymbol{\rho})] + \Phi(C_0, A_0).$$

□

In order to define the potential function, a matching is maintained between the on-line and the adversary servers. This matching is dynamic, i.e. it changes as the servers move. Initially, $C_0 = A_0$ and each on-line server is matched to the corresponding adversary server. When the adversary moves a server α to the request r_i (at the begining of the i-th phase) the matching does not change: the on-line server α was matched to before the move is the same that α is matched to after the move. When it is the turn of HARMONIC to service r_i there are two cases to consider. If the server h that HARMONIC will move to r_i is the on-line server matched to α then the matching remains unaltered: after its move, server h will still be matched to α. Otherwise, let s be the on-line server matched to α and β the adversary server h is matched to. After the move of server h, the matching changes: now h is matched to α and s to β. Nothing else changes.

Definition 3.6 *Let C, A be the current configurations of the on-line and the adversary servers respectively. Let B be any subset of A. Let h be any on-line server and call α the adversary server h is matched to.*

– *The radius $R(h,B)$ of B around h is defined as:*

$$R(h,B) = \max_{\beta \in B} d(h,\beta).$$

– *Let*

$$\Lambda(h) = k2^k \min_B \{ \frac{R(h,B)}{2^{|B|}} \mid \alpha \in B \}.$$

The active set AS(h) of h is defined to be the largest choice of B for which the value $\Lambda(h)$ is achieved. The notation S(h) will be used to indicate the size of the active set of h, i.e. $S(h) = |AS(h)|$.

Definition 3.7 (Potential function) *Let C, A be the current configurations of the on-line and the adversary servers respectively. The potential function describing the present system state is defined as:*

$$\Phi(C,A) = \sum_{h \in C} \Lambda(h)$$

.

Lemma 3.6 *The potential function satisfies property 2.*

Proof: The adversary will open the i-th phase of the game by selecting the next request point r_i and moving one of its servers there, thus changing its configuration from A_{i-1} to A_i. Let α be the adversary server chosen to service r_i. The cost $\text{cost}_{ADV}(r_i)$ that the adversary pays for serving request r_i is then $|A_{i-1}A_i|$. For any on-line server h let *AS(h)*, *S(h)* denote the active set of h and its size *before* the move of α. The move of the adversary server to r_i will, potentially, affect the value of the function $\Lambda(h)$ for some on-line servers h. Since we are interested in bounding the increase in the potential function resulting from the move of α it is enough to concentrate only on those on-line servers h for which $\Lambda(h)$ gets bigger. By the definition of the function Λ, all such servers must belong to the set

$$Y = \{h | \alpha \in AS(h) \text{ and } d(h, r_i) > R(h, AS(h))\}$$

Furthermore, for any $h \in Y$, the increase in $\Lambda(h)$ due to the move of the adversary server cannot be larger than $k2^k \frac{\text{cost}_{ADV}(r_i)}{2^{S(h)}}$ (this is due to the triangle inequality). Assume now that the on-line servers in Y are ordered, i.e. $Y = \{h_0, h_1, \ldots, h_{|Y|}\}$, so that

$$R(h_j, AS(h_j)) + d(h_j, r_i) \leq R(h_{j+1}, AS(h_{j+1}) + d(h_{j+1}, r_i), \qquad 1 \leq j \leq |Y| - 1$$

and h_0 is the on-line server matched to α, if that server belongs to Y. Otherwise there is no h_0 and the indexing in Y starts from h_1. It can then be shown that

$S(h_0) \geq 1$, $S(h_1) \geq 2$ and that $S(h_l) \geq l$, for all $l > 1$. Consequently:

$$\begin{aligned}
\Phi(C_{i-1}, A_i) - \Phi(C_{i-1}, A_{i-1}) &\leq \sum_{h \in Y} k2^k \frac{\mathrm{cost}_{ADV}(r_i)}{2^{S(h)}} \\
&\leq k2^k \mathrm{cost}_{ADV}(r_i)(\frac{1}{2} + \frac{1}{4} + \sum_{l=2}^{k-1} \frac{1}{2^l}) \\
&= (\frac{5}{4}k2^k - 2k)\mathrm{cost}_{ADV}(r_i).
\end{aligned}$$

which proves the lemma (with the factor c of property 2 being equal to $\frac{5}{4}k2^k - 2k$). $\square$

Lemma 3.7 *The potential function satisfies property 3.*

Proof: Assume that the HARMONIC algorithm is about to service request r_i which was just generated and served by the adversary. Let α be the adversary server covering r_i and s the on-line server matched to α just before HARMONIC makes its move. If h is any HARMONIC server then the probability of h being chosen to service r_i is equal to $\frac{1/d(h,r_i)}{N}$. Each such h belongs in one of three categories (in what follows AS and S refer to the matching just *before* the move of h):

1. $h = s$.

   In this case the move of the HARMONIC server affects only the value of $\Lambda(s)$, which drops to 0. For all other on-line servers x the value of $\Lambda(x)$ remains unchanged. Consequently, the contribution of this case to the the expected decrease of the potential is

   $$-\frac{\frac{1}{d(s,r_i)}}{N}\Lambda(s) = -k2^k \frac{R(s, AS(s))}{Nd(s, r_i)2^{S(s)}} \leq -\frac{k2^k}{N2^{S(s)}}.$$

2. *$h \neq s$ but the adversary server β matched to h is in AS(s).*

   Then, after the move of h, $\Lambda(h)$ becomes zero, $\Lambda(s)$ can only decrease while the value of Λ for all other on-line servers is not altered.

3. *$h \neq s$ and $\beta \notin AS(s)$ where β is the adversary server matched to h.*

   Again, after h moves, only $\Lambda(h)$ and $\Lambda(s)$ are affected, with the first dropping to zero. But $\Lambda(s)$ might now increase. As it turns out, this increase can be bounded. Let B be the set of all adversary servers that are outside *AS(s)* but within a distance of no more than $d(s, \beta)$ from s. Let $l = |B|$ and define *NS(s)* as

   $$NS(s) = AS(s) \cup B \cup AS(h).$$

Since $AS(s) \cap B = \emptyset$, $|NS(s)| \geq \max\{S(s)+l, S(h)\}$. Furthermore, due to the triangle inequality, $R(s, NS(s)) \leq d(s, r_i) + d(r_i, h) + R(h, AS(h)) \leq R(s, AS(s)) + d(r_i, h) + R(h, AS(h))$. From the definition of the function Λ it can then be seen that the following is true for the value of $\Lambda(s)$ *after* the move of the on-line server h:

$$\Lambda(s) \leq k2^k \frac{R(s, NS(s))}{2^{|NS(s)|}} \leq k2^k \frac{R(s, AS(s)) + d(r_i, h) + R(h, AS(h))}{2^{\max\{S(s)+l, S(h)\}}}.$$

As a result the expected increase in the potential in this case will be no more than

$$\frac{\frac{1}{d(h,r_i)}}{N}(\text{change in the value of } \Lambda(s) + \text{change in the value of } \Lambda(h)) \leq \frac{k2^k}{N2^{S(s)+l}}$$

Note also that, since there are $k - S(s)$ on-line servers in this category, the above inequality will happen for all l between 1 and $k - S(s)$.

Puting together the contribution of all these cases and observing that $E[cost_{HAR}(r_i)] = k/N$ we get

$$\begin{aligned} & E[cost_{HAR}(r_i)] + E[\Phi(C_i, A_i) - \Phi(C_{i-1}, A_i)] \\ \leq \;& \frac{k}{N} - \frac{k2^k}{N2^{S(s)}} + \sum_{l=1}^{k-S(s)} \frac{k2^k}{N2^{S(s)+l}} \\ \leq \;& 0. \end{aligned}$$

□

Lemmas 3.5, 3.6 and 3.7 together with the fact that, by definition, Φ is non-negative imply that the HARMONIC algorithm is $(\frac{5}{4}k2^k - 2k)$-competitive against any on-line adversary.

Although this result is the best known up to date, it is speculated that the HARMONIC algorithm has a much better, polynomial in k, competitive ratio against adaptive on-line adversaries. This speculation was originally presented in [23] in the form of the so called *lazy adversary conjecture.* A *lazy adversary* is a special case of an adaptive on-line adversary which operates as follows: as long as there exist adversary servers that do not coincide with any on-line server, a lazy adversary will pick for the next request a point occupied by such a server. The advantage of doing so is that the adversary can serve such a request at no cost (it already has a server there) while the on-line algorithm will have to move some server at the request point, thus incurring a non-zero cost. The speculation is that this strategy maximizes the ratio of the HARMONIC cost to the adversary cost.

Lazy adversary conjecture: *The worst possible competitive ratio for the HARMONIC algorithm (against an adaptive on-line adversary) is forced by some lazy adversary.*

It has been shown ([20], [23]) that the competitive ratio of the HARMONIC algorithm against any lazy adversary is no worse than $k(k+1)/2$ (in fact, it has also been shown that it is no better than that, i.e. there exist lazy adversaries forcing that competitive ratio). If the lazy adversary conjecture is true, then this would prove that the competitive ratio of HARMONIC against any adaptive on-line adversary is exactly $k(k+1)/2$ (no better and no worse).

4 The uniform metric space

A symmetric space (Q, d) is called *uniform* if all distances in the space are equal, i.e. for some $a > 0$, $d(x, y) = a$ for all pairs of distinct points $x, y \in Q$. The usual convention when talking about a uniform space is to assume that $a = 1$. This assumption has no consequence as far as competitive analysis is concerned. An algorithm that is shown to be c-competitive in a uniform space where $a = 1$ is also c-competitive in any other uniform space, as can be readily verified from Definitions 2.1 and 2.2. In what follows it is assumed that all distances in a uniform metric space are equal to one.

Another convention (which we also follow) is to discuss the on-line k-server problem for uniform spaces in terms of the caching problem (the relation between the two was exhibited at the end of Section 2). Instead of k servers in a uniform space of size n we will consider a cache of k page frames and a main memory of n pages. Every miss will carry a unit cost.

Due to the importance of the caching problem a considerable number of algorithms have appeared in the literature. In this section six of them are presented: three deterministic (LRU, FIFO, ROTATE) and three randomized (RANDOM, MARKING, PARTITIONING). One out of each category is memoryless (ROTATE, RANDOM). The performance of RANDOM is studied against any adaptive on-line adversary while for MARKING and PARTITIONING the adversary is an oblivious one. It will be shown that all the above algorithms, except MARKING, are *strongly competitive*, i.e. they achieve the best possible competitive ratio. For the deterministic algorithms this ratio is k, as already mentioned. In the case of randomized algorithms, Fiat at al. in [11] show that no randomized on-line algorithm for the caching problem can be better than H_k-competitive against any oblivious adversary (where $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$). Furthermore, for the special case of memoryless randomized algorithms, Raghavan and Snir ([23]) proved that their competitive ratio against adaptive on-line adversaries cannot be better than k.

4.1 LRU and FIFO

The *Least Recently Used (LRU)* and *First In First Out (FIFO)* page replacement algorithms were among the first heuristics used to deal with the caching problem. They were also among the first on-line algorithms to be analyzed in the context of competitive analysis. Sleator and Tarjan ([24]) were able to exhibit that both algorithms are strongly competitive.

Lemma 4.1 *LRU is k-competitive.*

Proof: Let OPT be the optimal off-line algorithm for the caching problem and $\boldsymbol{\rho}$ any sequence of requests. The aim is to show that for every set of k page faults forced by $\boldsymbol{\rho}$ on LRU, the optimal off-line algorithm suffers at least one page fault.

Assume that LRU and OPT are run with the request sequence $\boldsymbol{\rho}$ as input. Let $\boldsymbol{\rho}= \boldsymbol{\rho}_0\boldsymbol{\rho}_1 \ldots \boldsymbol{\rho}_m$ be a partitioning of $\boldsymbol{\rho}$ into subsequnces so that the number of page faults that LRU suffers during handling $\boldsymbol{\rho}_i$, $1 \leq i \leq m$ is exactly k (for $\boldsymbol{\rho}_0$ LRU suffers at most $(k-1)$ page faults). Let p_{i-1} be the page reference immediately before $\boldsymbol{\rho}_i$ (i.e. p_{i-1} is the last reference of $\boldsymbol{\rho}_{i-1}$). Examining the faulting behaviour of LRU in $\boldsymbol{\rho}_i$ we can distinguish two cases. The first one is when all k faulting pages in $\boldsymbol{\rho}_i$ are distinct and none of them is p_{i-1}. Since p_{i-1} is in the cache of OPT just before $\boldsymbol{\rho}_i$, at least one of these pages will force a page fault on OPT. The second case is when p_{i-1} is among the faulting pages or when there is some page creating two or more page faults. By the definition of LRU, this means that there are at least $(k+1)$ distinct pages in $\boldsymbol{\rho}_i$. As a result, OPT will fault on at least one of them.

In either case each $\boldsymbol{\rho}_i$, $1 \leq i \leq n$, forces one or more page faults on OPT. Taking also into account the page faults that LRU suffers due to $\boldsymbol{\rho}_0$ we can conclude that

$$\text{cost}_{LRU}(\boldsymbol{\rho}) \leq k \cdot \text{cost}_{OPT}(\boldsymbol{\rho}) + (k-1)$$

which proves the claim. □

Using a similar argument, FIFO can also be shown to be k-competitive.

4.2 ROTATE

ROTATE ([4]) is a simple memoryless algorithm defined as follows:

Definition 4.1 (ROTATE) *Each time that a miss occurs on a page reference r evict the cache page s, where*

$$s = \max_{\textit{cache pages } p} \{p \mid p < r\}$$

If there is no page number less than r in the cache, then evict the page with the greatest page number.

The definition assumes that pages are identified by page numbers (which is usually the case) so that the comparisons are meaningful. The algorithm can be extended, though, to any uniform space. All that is required is an (arbitrary) total ordering on the points of the space.

Lemma 4.2 *ROTATE is k-competitive.*

Proof: The proof is similar to the one used to prove HARMONIC competitive. The algorithm is assumed to be playing against an adversary which generates and immediately serves the requests. A non-negative potential function is then defined and is shown to have the following properties: first, each time the adversary serves a request the increase in the potential function is at most k times the adversary cost for serving that request; second, each time ROTATE serves a request the potential drops by at least as much as the algorithm paid for servicing the request. Such a potential function will guarantee that ROTATE is k-competitive.

The potential function used is defined in terms of the current algorithm and adversary configurations (where, in the context of caching, a configuration is the set of pages in the cache). To help visualize the discussion assume that every page number r is identified with the point r of the real line. There are two sets $C = \{s_1, \ldots, s_k\}$ and $A = \{\alpha_1, \ldots, \alpha_k\}$ of k mobile servers each, where every server can occupy a point r of the real line corresponding to the page with the same number. The points covered by the *algorithm* servers of C represent the pages currently in the cache of ROTATE while the *adversary* servers of A cover the pages currently in the adversary's cache.

For any two numbers a, b define the interval $(a, b]$ as

$$(a,b] = \begin{cases} \{x \mid a < x \le b\} & \text{, if } a \le b \\ \{x \mid x > a \text{ or } x \le b\} & \text{, otherwise} \end{cases}$$

Let $v_{i,j}$ be the number of adversary servers in the interval $(s_i, \alpha_j]$ where, as usually, we abuse the notation by identifying a server with the number of the page that it covers. For any permutation π over the set $\{1, 2, \ldots, k\}$ define

$$v(\pi) = \sum_{i=1}^{k} v_{i,\pi(i)}.$$

The potential function Φ is then defined as

$$\Phi(C, A) = \min_{\pi} v(\pi).$$

A permutation π for which the minimum above is achieved is said to *realize* Φ.

In order to show that the Φ has the first of the two claimed properties, let r be the new request generated by the adversary. If $r \in A$ nothing happens and the property is trivially true. So, let $r \notin A$ and assume that it is the adversary server α_j that moves to r. Let π be a permutation that realizes Φ just before the move of α_j. Imagine that in order to serve r_i the server α_j starts moving rightwards on the real line (towards infinity) until it either hits r_i or the last page covered by a (adversary or algorithm) server. In the later case it "wraps around" to the origin of the real line and continues moving to the right. In either case, when α_j reaches r_i it stops. During this trip α_j will pass by a number of adversary and algorithm servers. When it passes by an algorithm server s_i the potential increases by one (since the interval $(s_i, \alpha_{\pi(i)}]$ now contains one more adversary server). When it crosses over an adversary server α_l the potential either drops by one (because the adversary server α_j just deserted the interval $(s_{\pi^{-1}(l)}, \alpha_l])$ or remains unchanged (because the drop is counterbalanced by the fact that α_l was added to the interval $(s_{\pi^{-1}(j)}, \alpha_j])$. In any case, the increase in the potential is no greater than k, the number of algorithm servers.

For the second property, let r be the new request that ROTATE must serve and assume that α_j is the adversary server covering r. Let Φ be the potential function just before ROTATE serves r and consider a permutation π realizing Φ such that for every $i \neq l$ none of the intervals $(s_i, \alpha_{\pi(i)}]$, $(s_l, \alpha_{\pi(l)}]$ is a subset of the other (there always exists such a π). If r is already covered by an algorithm server there is no change and the property is true. So assume that $r \notin C$. If $s_{\pi^{-1}(j)}$ is the algorithm server that ROTATE will pick up for servicing r, then the potential drops by at least one after the move (since α_j is no longer in the interval $(s_{\pi^{-1}(j)}, \alpha_j])$. Otherwise, let s_i be the ROTATE server that will move. Since s_i is the first algorithm server to the "left" of r, it follows that r (and thus α_j) belongs to the interval $(s_i, \alpha_{\pi(i)}]$ or else $(s_i, \alpha_{\pi(i)}]$ would be a subset of $(s_{\pi^{-1}(j)}, \alpha_j])$. Consequently, moving s_i to r drops the potential by at least one. □

4.3 MARKING

The MARKING algorithm, presented and analyzed by Fiat et al. in [11], is a randomized version of LRU. It operates by maintaining a dynamic set of marks, each one occupying some cache frame. In case of a miss the cache page to be evicted is selected at random among the unmarked ones. More specifically:

Definition 4.2 (MARKING) *Initially, all cache page frames are unmarked. Let r be a new page reference. If r is already in the cache, then a mark is placed on the corresponding cache frame (if one is not already there). Otherwise, the victim page is chosen uniformly at random among those that do not have a mark. If all pages in the cache are marked, then before the random selection of the victim all the marks are cleared. In either case, at the end a mark is placed on the cache frame where r is stored.*

Lemma 4.3 *The MARKING algorithm is $2H_k$-competitive against any oblivious adversary.*

Proof: The algorithm is assumed to be playing against an oblivious adversary which generates the input page reference sequence $\boldsymbol{\rho} = (r_1, \ldots, r_m)$. Initially the algorithm configuration C_0 and the adversary configuration A_0 are the same. Let $\boldsymbol{\rho} = \boldsymbol{\rho}_0 \boldsymbol{\rho}_1 \ldots \boldsymbol{\rho}_t)$ be a partitioning of $\boldsymbol{\rho}$ where $\boldsymbol{\rho}_0$ only contains pages in C_0, $\boldsymbol{\rho}_1$ starts with the first page reference not in C_0 and each $\boldsymbol{\rho}_i$, $1 \leq i < t$, is a maximum length subsequence containing k *distinct* pages (i.e. if r is the first reference in $\boldsymbol{\rho}_{i+1}$ then the subsequence $\boldsymbol{\rho}_i r$ contains exactly $(k+1)$ distinct pages). The only exception is $\boldsymbol{\rho}_t$ since there might not be enough references at the end of $\boldsymbol{\rho}$ to guarantee that $\boldsymbol{\rho}_t$ will have k distinct pages.

Let $r_j \in \boldsymbol{\rho}_i$ be a reference to a page not appearing previously in $\boldsymbol{\rho}_i$. Then r_j is called *clean* if there is no reference to the same page in $\boldsymbol{\rho}_{i-1}$ while it is called *stale* otherwise. Let C_i, A_i be the configurations of the algorithm and the adversary respectively immediately after the last reference of $\boldsymbol{\rho}_i$. From the definition of MARKING, C_i contains exactly the pages referenced in $\boldsymbol{\rho}_i$. Let l_i be the number of clean vertices in $\boldsymbol{\rho}_i$ and define $d_i = |A_i - C_i|$. That is, d_i indicates the number of pages which the adversary and algorithm caches differ at after finished processing $\boldsymbol{\rho}_i$. Under the above definitions it is not hard to verify that

$$\text{cost}_{ADV}(\boldsymbol{\rho}_i) \geq (l_i - d_{i-1}) \quad \text{and} \quad \text{cost}_{ADV}(\boldsymbol{\rho}_i) \geq d_i$$

which imply that $\text{cost}_{ADV}(\boldsymbol{\rho}_i) \geq \frac{1}{2}(l_i - d_{i-1} + d_i)$. Summing over all i:

$$\text{cost}_{ADV}(\boldsymbol{\rho}) \geq \frac{1}{2}\sum_{i=1}^{t} l_i.$$

The expected cost of MARKING on $\boldsymbol{\rho}_i$ has two components. The first one is due to serving the clean references. Every such reference imposes a unit cost since it always creates a miss. The second component is the expected cost of servicing the stale references. It can be shown that this cost is never greater than $l_i(H_k - H_l)$. Combining the contribution of both clean and stale page references

$$E[\text{cost}_{MAR}(\boldsymbol{\rho}_i)] \leq l_i(H_k - H_l + 1) \leq H_k l_i.$$

Summing over all i:

$$E[\text{cost}_{MAR}(\boldsymbol{\rho})] \leq H_k \sum_{i=1}^{t} l_i \leq 2H_k \text{cost}_{ADV}(\boldsymbol{\rho}).$$

□

MARKING was the first randomized on-line algorithm for the k-server problem in the uniform space proved to have a sub-linear competitive ratio, thus exhibiting that, when playing against an oblivious adversary, randomized algorithms have an edge over deterministic ones. The advantage of a good randomized algorithm is that, by being able to respond in many possible ways to any given request sequence $\boldsymbol{\rho}$, it can balance (in a probabilistic sense) high and low cost responses thus producing a favorable overall expected cost.

4.4 PARTITIONING

The PARTITIONING algorithm, due to McGeogh and Sleator ([19]), is another randomized version of LRU which, however, is provably optimal: PARTITIONING is H_k-competitive against any oblivious adversary. It is, however, substantially more complicated than the MARKING algorithm. In what follows a description of the algorithm is given together with an outline of its competitive analysis. As usual, OPT will stand for the optimal off-line algorithm.

The algorithm is based on a partition of all the pages in the memory space into an ordered sequence of disjoint sets $S_a, S_{a+1}, \ldots, S_{b-1}, S_b$ (some of which might be empty) where $a < b$. This partition is dynamic; it is updated (in a way to be described shortly) each time a new page reference arrives. With each set S_i, $a \le i < b$, we associate a non-negative label l_i. The labels are related to the cardinalities of the sets and always satisfy the following *labaling invariant*:

$$\begin{aligned} &l_a = 0 \\ &l_i = l_{i-1} + |S_i| - 1 \qquad (a < i < b) \\ &l_{b-1} = k - |S_b|. \end{aligned}$$

Initially $a = 1$, $b = 2$ and there are only two sets in the partition. The set S_2 contains the k pages originally residing in the cache of the algorithm and the set S_1 the remaining $n - k$ pages (where n is the size of the memory space). When a new page reference r arrives the partitioning is updated in the following way:

$r \in S_b$: No change.

$r \in S_i, a < i < b$: First the following assignments are applied: $S_i = S_i - \{r\}$, $S_b = S_b \cup \{r\}$ and $l_j = l_j - 1$ (for $i \le j < b$). If after these assignments there exist some label that is equal to zero, let j be the greatest integer for which $l_j = 0$. Then the partition is modified as follows:

$$S_a = \bigcup_{t=1}^{j} S_t \qquad \text{and} \qquad a = j.$$

$r \in S_a$: The following assignments are performed: $S_a = S_a - \{r\}$, $S_{b+1} = \{r\}$, $l_b = k - 1$ and $b = b + 1$.

It is not hard to verify that this updating process maintains the labeling invariant.

Assume that $\boldsymbol{\rho} = (r_1, \ldots, r_m)$ is the input page reference sequence and let $P_i = \{S_a, \ldots, S_b\}$ be the partition immediately after the page reference r_i has been processed. The purpose of P_i is to provide information about the contents of OPT's cache after OPT has served r_i. These contents are of course dependent on input yet unseen (the requests $r_{i+1}, \ldots, r_m$). Still, it is possible to have a partial idea about what pages the cache of OPT contains. For example, at the very least we know that it includes the page r_i. In a less trivial situation assume that $(r', r, \ldots)$ is the input reference sequence and that the cache of OPT has two frames that initially contain the pages r and r''. When the first reference r' arrives we know that it creates a miss but it is not possible to infer which of its two cache pages OPT is going to evict. When r arrives, though, it can be safely inferred that it was r'' that OPT evicted at the previous miss. So, after OPT processes the second reference r, its complete cache image is known, even without information about future requests. It is this kind of reasoning that the partition is supposed to capture. More specifically, let $P_0 = \{S_1, S_2\}$ be the initial partition and consider the execution of OPT on the sequence $\boldsymbol{\rho}$ when started with its cache containing the set of pages S_2. Then the partition P_i $(0 \leq i \leq m)$ provides the following information about the cache image of OPT immediately after OPT has served r_i:

1. If $p \in S_b$ then the page p is in the cache of OPT.

2. If $p \in S_a$ then p in not in the cache of OPT.

3. If $p \in S_j$, $a < j < b$ then no definite conclusion about the state of p can be drawn based on what has been seen up to now (i.e. p might or might not be in OPT's cache, depending on future requests).

4. Every set S_j $(a < j < b)$, cannot have more than $l_j^* = \min_{j \leq t < b}\{l_t\}$ pages in OPT's cache.

Furthermore, a request r_i will create a miss for OPT if and only if r_i belongs to the set S_a of the partition P_{i-1}.

The PARTITIONING algorithm operates by maintaining the dynamic partition described above augmented with a set of marks. There is a distinct kind of mark (called an *i-mark*) for every set $S_i^* = \bigcup_{a \leq j \leq i} S_j$ $(a < i < b)$ and there are l_i copies of each i-mark. An i-mark can only be placed on a page in S_i or on a page already possesing an $(i-1)$-mark. Since $l_i = l_{i-1} + |S_i| - 1$, there are $l_i + 1$ potential recipients of an i-mark.

The algorithm always keeps in its cache the pages of S_b (i.e. the pages that are for sure in OPT's cache) as well as all pages bearing a $(b-1)$-mark. Initially $a = 1$, $b = 2$ and there are no marks.

Definition 4.3 (PARTITIONING) *Let r be an incoming page reference and $P = \{S_a, \ldots, S_b\}$ the partition just before the arrival of r. If r is already in the cache (i.e. $r \in S_b$ or r has a (b–1)-mark in P) there is nothing to do. Otherwise there are two cases to consider:*

1. *If $r \in S_i$, $a < i < b$, then* before *updating the partition perform the following step for each j such that $i \leq j < b$:*

If r has a j-mark then do nothing. Otherwise choose uniformly at random a page p that has a j-mark and transfer each t-mark (for $t \geq j$) of p to r. If p has a (b–1)-mark, then p is the page to be evicted from the cache.

Subsequently update the partition and remove all marks from r (which by now is in S_b). If in the process of updating the partition the set S_a changes then remove all marks from the pages in the new S_a.

2. *If $r \in S_a$ then first update the partition thus creating a new S_b containing only the page r. Then create $(k-1)$ new (b–1)-marks and distribute them uniformly at random among the k pages eligible of receiving such a mark (these are exactly the pages in the cache of the algorithm just before the arrival of r). The one page that does not receive a (b–1)-mark is the one that is evicted.*

Since, as mentioned above, there are (l_i+1) possible recipients of an i-mark, the total number of valid mark arrangements is $\prod_{a<i<b}(l_i+1)$. It can be shown that the PARTITIONING algorithm is equally likely to produce any of these valid arrangements. Using this fact it is possible to prove the following:

Lemma 4.4 *Let $r \in S_i$, $a < i < b$, be an incoming page reference. The probability of r creating a miss is no greater than*

$$\sum_{i \leq j < b} \frac{1}{l_j+1}.$$

This lemma is used in proving that

Lemma 4.5 *The PARTITIONING algorithm is H_k-competitive against any oblivious adversary.*

Proof sketch: Let $\boldsymbol{\rho} = (r_1, \ldots, r_m)$ be the input sequence generated by the adversary (it can be assumed that the adversary will serve $\boldsymbol{\rho}$ optimally) and define the non-negative potential function Φ as

$$\Phi(P_i) = \sum_{a \leq i < b} (H_{l_i+1} - 1)$$

where P_i is the partition immediately after r_i has been processed. It can then be shown, using induction on the lenght of $\boldsymbol{\rho}$, that the following inequality is true

$$\mathrm{cost}_{PAR}(\boldsymbol{\rho}) + \Phi(P_m) \leq H_k \cdot \mathrm{cost}_{OPT}(\boldsymbol{\rho})$$

which proves the lemma. The induction proceeds by assuming that this inequality holds before the arrival of the last page reference r_m and shows that it also holds after r_m has been processed. There are two cases to consider. The first is when r_m is in some set S_i $(a < i < b)$ of the partition P_{m-1} (at which case r_m is a hit for OPT) while the second is when $r_m \in S_a$ (a miss for both OPT and PARTITIONING). Lemma 4.4 provides a bound for the cost of PARTITIONING in the first case. □

4.5 RANDOM

RANDOM, a simple memoryless randomized algorithm introduced by Raghavan and Snir in [23], is really the HARMONIC algorithm restricted to the uniform space:

Definition 4.4 (RANDOM) *Let r be an incoming page reference. If r creates a miss, then choose a cache page uniformly at random and evict it.*

Lemma 4.6 *RANDOM is k-competitive against any adaptive on-line adversary.*

Proof: The proof is similar to the one used for proving HARMONIC competitive. RANDOM is assumed to be playing against an adaptive on-line adversary that generates the input page reference sequence $\boldsymbol{\rho} = (r_1, \ldots, r_m)$. Let C_i (A_i) indicate the set of pages in the cache of RANDOM (the adversary) immediately after the algorithm (the adversary) has served r_i. Initially $C_0 = A_0$. Define the non-negative potential function Φ as

$$\Phi(C, A) = k(k - |C \cap A|)$$

where C, A are the current cache contents of RANDOM and the adversary respectively. Consider the change in the potential when the adversary generates the next request r_i. If $r_i \in A_{i-1}$ then $A_i = A_{i-1}$ and $\text{cost}_{ADV}(r_i) = 0$. If $r_i \notin A_{i-1}$ then the adversary will choose one of its cache pages and will evict it, thus suffering a cost equal to one and changing its cache image from A_{i-1} to A_i. In either case

$$\Phi(C_{i-1}, A_i) - \Phi(C_{i-1}, A_{i-1}) \leq k \cdot \text{cost}_{ADV}(r_i).$$

Consider now what happens when RANDOM is about to serve the request r_i which was just generated and served by the adversary. If r_i is a hit for RANDOM then no cost is paid for serving it and there is no change in the value of the potential. Otherwise a cache page must be evicted. There are $(k - |A_i \cap C_{i-1}|) \geq 1$ pages in RANDOM's cache that are not in the cache of the adversary and each one of them has probability $\frac{1}{k}$ of being evicted. The eviction of any such page causes an expected drop of -1 to the value of the potential function while evicting any page from $C_{i-1} \cap A_i$ leaves the potential unchanged. Consequently, no matter what r_i is for RANDOM (a hit or a miss) we have that

$$E[cost_{RAN}(r_i)] + E[\Phi(C_i, A_i) - \Phi(C_{i-1}, A_i)] \leq 0$$

The combined effect of the above two inequalities (as exhibited in the analysis of HARMONIC) is that RANDOM is k-competitive against any adaptive on-line adversary. □

As mentioned at the begining of the section, no memoryless randomized algorithm can be better than k-competitive against oblivious adversaries and RANDOM was just shown to achieve that optimal behaviour. This means that when no information about the past is available, the strategy of randomly choosing a victim in the case of a miss is the best.

5 Trees

Trees are yet another category of symmetric spaces.

Definition 5.1 *A metric space (Q, d) of n points is called a tree if there exists an undirected weighted tree $T = (V, E)$, $|V| \geq n$, with non-negative edge costs and a one-to-one mapping $f : Q \to V$ such that for every pair of points $x, y \in Q$ the distance $d(x, y)$ is equal to the weight of the unique path in T connecting the vertices $f(x)$ and $f(y)$.*

The family of trees contains several interesting spaces, although their tree structure might not be immediately recognizable. As an example consider the uniform space of n points. This space can be mapped on a star-shaped tree with $(n + 1)$ vertices. Each point of the space corresponds to a leaf and the distance of every leaf from the center of the star is equal to $\frac{1}{2}$.

Chrobak and Larmore in [5] present a deterministic on-line k-server algorithm for trees and show that it is optimal. An interesting feature of the algorithm is that when serving a request it can move around more than one servers (none of the algorithms presented so far had this property). In the following discussion it is assumed that the mapping of the underlying space to a weighted tree T has been established (as prescribed by the definition above) and we imagine all the action taking place on that tree.

Definition 5.2 (The algorithm) *Let $C = \{s_1, \ldots, s_k\}$ be the current configuration of the k servers, where each s_i is a vertex of T. Let r be an incoming request. Define $(x, y]$, for any two points x, y of T to be the unique path connecting x and y but with the point x excluded. A server $s_i \in C$ will be called active (relative to r) if there exists no other server on the path $(s_i, r]$. If it happens that more than one servers currently occupy the point s_i then one is chosen arbitrarily and deemed to be the active server. All the others are inactive. In order to serve r, the following loop is executed:*

while *none of the algorithm's servers is on* r **do**

- *Let* $\delta = \min_s\{d(s, y_s) \mid s$ *is an active server*$\}$ *where* y_s *is the first vertex in* $(s, r]$.
- *Move each active vertex a distance* δ *towards* r.

endwhile

Notice that a server s that begins as active might be rendered inactive midway through its journey to r. This will happen if during the execution of the loop described above some other active server appear in front of s.

Lemma 5.1 *The algotithm is* k*-competitive.*

Proof: The standard approach is followed: the algorithm is playing against an adversary that generates the input request sequence $\boldsymbol{\rho} = (r_1, \ldots, r_m)$. As usually, C_i and A_i denote the configurations of the algorithm and the adversary respectively immediately after they serve r_i. A non-negative potential function is defined. When the adversary moves to serve a request the potential will not increase by more than k times the adversary cost. When the algorithm moves the potential will decrease by at least the cost paid by the algorithm. These conditions guarantee the k-competitiveness of the algorithm. The potential function Φ is defined as:

$$\Phi(C, A) = k \cdot |CA| + \sum_{i<j} d(s_i, s_j)$$

where $C = \{s_1, \ldots, s_k\}$ and $A = \{\alpha_1, \ldots, \alpha_k\}$ are the current configurations of the algorithm and the adversary and $|CA|$ is the minimum matching distance between C and A.

Assume that the adversary generates the request r_i and moves one of its servers, say α_j, to serve it thus paying a cost $d = d(\alpha_j, r_i)$ and changing its configuration from A_{i-1} to A_i. The summation component of the potential is not affected, since no on-line servers are moved, while the matcing distance does not increase more than d due to the triangle inequality. As result, the increase in the potential is no more than $k \cdot d$.

Consider now the move of the algorithm to serve r_i (which is by now covered by the adversary server α_j). Without loss of generality we can assume that $s_1, \ldots, s_t$ are the active (with respect to r_i) on-line servers. It can now be seen that there is always a minimum matching between C_{i-1} and A_i such that the adversary server α_j is mapped to some active on-line server. Assume that s_1 is that server. Consider the first execution of the algorithm's while-loop. All active servers will move some distance δ for an overall cost of $t\delta$. During this move, the minimum matching distance cannot increase by more than $(t-2)\delta$ (s_1, at the very least, moves closer to its matched adversary server). For $p = 1, \ldots, t$

let l_p be the number of inactive servers s such that s_p is in $(s, r_i]$ (for each such s there is exactly one s_p with the above property and as a result $\sum_{p=1}^{t} l_p = k-t$). As s_p moves by δ, its distance to these l_p on-line servers increases by δ but to the remaining $(k-l_p-1)$ servers its distance decreases by δ. Putting everything together we conclude that the change in the potential is no more than

$$k(t-2)\delta + \sum_{p=1}^{t}(l_p - k + l_p + 1)\delta = -t\delta$$

which shows that the potential drop covers for the cost of the algorithm in the first execution of the while-loop. Exactly the same reasoning applies for subsequent iterations of the loop. □

6 Resistive spaces

In designing a randomized algorithm for the on-line k-server problem, the primary consideration is how to compute the probability distribution of the possible responses to an incoming request. HARMONIC, for example, advocates that any server should move to the request with a probability inversely proportional to its distance from it. Coppersmith et al. ([7]) propose a different approach for the case when the underlying space is *resistive*.

Definition 6.1 *A metric space (Q, d) of n points $q_1, \ldots, q_n$, is called resistive if there exists a network of resistors N_Q with n nodes $v_1, \ldots, v_n$ such that for every i, j $(1 \leq i, j \leq n)$*

$$d(q_i, q_j) = R_{v_i, v_j}$$

where $R_{v,u}$ is the effective resistance between the nodes v and u in N_Q. The conductance matrix $\{\sigma_{v,u}\}$ of N_Q (where $1/\sigma_{v,u}$ is the branch resistance between the nodes u and v) is called the resistive inverse of (Q, d).

In a related definition, a metric space is called *m-resistive* if every m-subset of the space is resistive. It can be shown that every symmetric space is 3-resistive. Furthermore, every subset of a resistive space is also resistive. Consequently every resistive space with n points is m-resistive for every $m \leq n$.

It turns out that resistive spaces are a quite broad class including, among others, all uniform spaces and all tree spaces. Coppersmith et al. in [7] introduce RWALK, a randomized and memoryless k-server algorithm, for the case when the underlying space is $(k+1)$-resistive. In doing so they use the resistor network underlying any $(k+1)$-subset of the space as a guide for computing the probability of each server moving to serve a request.

Definition 6.2 (RWALK) *Let $C = \{s_1, \ldots, s_k\}$ be the current configuration of the servers when the request r arrives. If $r \in C$ then the request point is* *already covered and is served at no cost. Otherwise let $\{\sigma_{v,u}\}$ be the resistive inverse of the subspace $C \cup \{r\}$. Then the probability of the server s_i servicing r is*

$$\frac{\sigma_{s_i,r}}{\sum_{j=1}^{k} \sigma_{s_j,r}}.$$

Lemma 6.1 *RWALK is k-competitive against any adaptive on-line adversary.*

Proof sketch: The standard method is employed in this proof too. The algorithm is assumed to be playing against an adaptive on-line adversary that generates the input request sequence. A non-negative potential function is defined and is shown to have the properties 2 and 3 described in Section 3.2 (with the factor c of property 2 being equal to k). In fact the potential function used here is exactly the same as the one used for proving the tree algorithm of the previous section competitive (it was subsequently shown by Deng and Mahajan ([8]) that this potential function has the afore mentioned properties for a randomized and memoryless on-line algorithm if and only if the underlying space is $(k+1)$-resistive). Proving that this potential function respects property 2 is quite straightforward: the same argument used in the proof of the tree algorithm is used here too. For property 3 the argument is a little more technical but still not complicated. It uses the mathematical formulation for defining an m-resistive space. □

Since, as mentioned above, every symmetric metric space is 3-resistive, RWALK gives a 2-competitive algorithm for the 2-server problem on every such space. RWALK can also be used to provide a competitive algorithm for an unrestricted space, given that this space has a resistive approximation.

Definition 6.3 *A resistive space (Q, d') is said to be a resistive λ-approximation of the metric space (Q, d) (where $\lambda > 0$) if for every $p, q \in Q$:*

$$d'(p,q) \leq d(p,q) \leq \lambda d'(p,q)$$

Lemma 6.2 *If a metric space (Q, d) has a resistive λ-approximation (Q, d') then there exists a randomized and memoryless k-server algorithm for (Q, d) that is λk-competitive against any adaptive on-line adversary.*

Proof: Assume that the input request sequence $\boldsymbol{\rho} = (r_1, \ldots, r_m)$ is served by RWALK in (Q, d'). Every sequence of server moves produced by RWALK in serving $\boldsymbol{\rho}$ has a d-cost that is no greater than λ times the corresponding d'-cost. Taking expectations the result follows. □

In fact the above reasoning applies for other kinds of approximation as well. For example, if all distances of (Q, d) are in the interval $[1, \lambda]$ (or, more generally, if $\max_{p,q \in Q} d(p,q) \leq \lambda \min_{p,q \in Q} d(p,q)$) then any of the k-competitive uniform space algorithms can be used to give a λk-competitive algorithm for (Q, d).

7 Weighted cache

In this section we conclude the presentation of on-line algorithms for the k-server problem by considering an instance of the problem in an asymmetric metric space. Let (Q, d) be a space where every point $q \in Q$ has an associated positive weight $w(q)$ and where the distance $d(p, q)$ of point q *from* any other point $p \in Q$ is defined as

$$d(p, q) = w(q).$$

It is not hard to see that this distance function satisfies the triangle inequality property and, as a result, (Q, d) is a metric space. But $d(p, q) = w(q)$ which, in general, is different from $w(p) = d(q, p)$. The k-server problem over such a space is known as the *weighted cache* problem since it can be considered a generalized version of the cache problem where the cost of a miss is not uniform but depends on the page that created the miss. Caching fonts in a printer or in the memory of a bitmap display are examples of weighted cache problems.

Raghavan and Snir ([23]) discuss a version of the HARMONIC algorithm for the weighted cache problem. In this setting, a server that currently occupies a point s will move to serve an incoming request r with probability inversely proportional to $w(s)$. The algorithm is proven k-competitive against any adaptive on-line adversary when the maximum weight in the underlying space is finite. Another algorithm, deterministic this time, is described by Chrobak et al. in ([4]). The algorithm, called BALANCE, is shown to be k-competitive. The remaining of this section contains a description of this algorithm and an outline of its competitive analysis.

BALANCE operates by trying to equally divide the service cost of a request sequence among all the k servers. To this end each server maintains a variable that stores the work performed by the server so far. The value of this variable is initialized to zero and is updated each time the server is selected to service a request.

Definition 7.1 (BALANCE) *Let the algorithm servers be labeled $s_1, \ldots, s_k$ arbitrarily and with each s_i associate a work variable W_i initialized to zero. Each time a request r arrives one of two things can happen: either the request point r is already covered by a server, at which case it is served by that server at zero cost, or not. If r is uncovered then a server s_j, such that $W_j = min\{W_1, \ldots, W_k\}$, is chosen and is moved to r. Finally, W_j is incresed by $w(r)$, the cost of serving the request.*

Lemma 7.1 *BALANCE is k-competitive.*

Proof: As usually, the algorithm is assumed to be playing against an adversary that generates the input request sequence $\boldsymbol{\rho} = (r_1, \ldots, r_m)$. Without loss of generality the following assumptions are made:

1. The adversary will never generate a request at a point currently occupied by an on-line server — omitting such a request leaves the cost of BALANCE unchanged while, due to the triangle inequality, never increases the adverary cost.

2. After an adversary server moves from a point q there is no subsequent request to q (such a request can be placed instead to an imaginary point q' that has the same weight as q).

3. At the end both the adversary and the algorithm servers are at the same configuration. The algorithm servers can be forced to converge to the final adversary configuration A_f by extending $\boldsymbol{\rho}$ with a sufficient number of requests to points of A_f currently unoccupied by the algorithm servers. The exessive requests will cost the adversary nothing while they can only increase the total cost of the algorithm. We assume that these extra requests are part of the sequence $\boldsymbol{\rho}$.

The set of points $R = \{r | \, r \in \boldsymbol{\rho}\}$ can now be partitioned in two disjoint sets: the set of final points $F = R \cap A_f$ and the set of non-final points $\mathrm{NF} = R - F$. Let W_F, W_{NF} be the sum of the weights of the final and non-final points respectively. Then, according to the above mentioned assumptions, we have that

$$\mathrm{cost}_{ADV}(\boldsymbol{\rho}) = W_{NF} + W_F.$$

Let $T(j)$ always denote the value of $\min\{W_1, \ldots, W_k\}$ just *before* the arrival of request r_j. Let $W_i(j)$ be the value of W_i just *after* the request r_j has been served. Clearly $T(0) = 0$ and $T(j)$ will remain zero as long as there exists an on-line server that has not still moved from its initial position. Consider now a server s_i that moves to serve a request $r_j \in \mathrm{NF}$. Since r_j is a non-final point, s_i will, at some point in the future, have to vacate r_j. Let r_l $(l > j)$ be the next request that makes s_i move from r_j. According to the definitions given above we have that

$$T(j) = W_i(j) - w(r_j) \qquad \text{and} \qquad T(l) = W_i(j).$$

Thus, during the time that s_i occupies the point r_j the value of T increases by $w(r_j)$. Using this fact it can be shown that

$$T(m) \leq W_{NF}.$$

To see this, assume that every request is first served by the adversary and then by the algorithm. Consider the case where an on-line server s_i moving to service a request r_j causes an increase in the value of T (i.e. $T(j) > T(j-1)$). Since the adversary has already moved a server at r_j, there will always be, just before the move of s_i, at least one (necessarily non-final) point q occupied only by an on-line server s_l (perhaps $s_l = s_i$). The increase of T is then charged to that point. But while q is occupied by s_l the increase in T is $w(q)$. As a result the total charge allocated to q cannot be more than $w(q)$. Consequently, $T(m) \leq \sum_{q \in \mathrm{NF}} w(q) = W_{NF}$.

The definition of the function T now implies that $T(j) \geq W_i(j) - w(q_i)$, where q_i is the point occupied by s_i immediately after r_j has been served by the algorithm. Putting everything together:

$$\mathrm{cost}_{BAL}(\boldsymbol{\rho}) = \sum_{i=1}^{k} W_i(m) \leq kT(m) + W_F \leq k(W_{NF} + W_F) \leq k \cdot \mathrm{cost}_{ADV}(\boldsymbol{\rho})$$

□

8 Lower bounds

The question of how well on-line k-server algorithms can perform has been addressed up to now from a "positive" side, by presenting concrete examples of such algorithms and analyzing their performance in the framework of competitive analysis. The same question can also been approached from the "negative" viewpoint of lower bounds: showing what cannot be achieved by an on-line algorithm. Some of these negative results have already been introduced in the discussion of the k-server problem over unrestricted and uniform spaces. This section further elaborates on these results and also introduces a few more of the same negative flavor.

The first lemma presented here appeared in the work of Manasse et al. ([18]) where the k-server problem itself was introduced. It gives a lower bound for the competitive ratio achievable by deterministic algorithms over symmetric spaces. This bound is strongly conjectured to be tight. The task of settling this conjecture has been the centerpiece of the research on the on-line k-server problem.

Lemma 8.1 *No deterministic algorithm for an instance of the on-line k-server problem over a symmetric metric space can be better than k-competitive.*

Proof: Consider an instance of the k-server problem on some symmetric space (Q, d). Let L be an on-line algorithm for that instance of the problem and let $C_0 = \{s_1, \ldots, s_k\}$ be an arbitrary initial configuration of L. Without loss of generality it can be assumed that C_0 contains k distinct points, i.e. no two servers of L coincide. We can also assume that L is "reasonable", meaning that it will only move a server in order to serve a request at an uncovered point (the triangle inequality inplies that doing otherwise can only increase the cost that L pays in processing a request sequence). Consider now any subspace H of (Q, d) containing $(k+1)$ points: the k points of C_0 plus an arbitrary point $q \in Q$. Then a request sequence $\boldsymbol{\rho} = (r_1, \ldots, r_m)$ of an arbitrary length m can be formed by always placing the next request at the unique point of H that is not covered by a server of L. So, $r_1 = q$, $r_2 = s_i$ where s_i is the point of C_0 that was occupied by the server that serviced r_1 and so on. This scheme then implies that the server which serves request r_i was occupying, just before the arrival of r_i, the point r_{i+1}. As a consequence, the cost paid by L in processing $\boldsymbol{\rho}$ is:

$$\mathrm{cost}_L(\boldsymbol{\rho}) = \sum_{i=1}^{m-1} d(r_{i+1}, r_i) + \mathrm{cost}_L(r_m) \geq \sum_{i=1}^{m-1} d(r_i, r_{i+1})$$

Consider now a family $B_1, \ldots, B_k$ of k server algorithms that are going to serve the same request sequence $\boldsymbol{\rho}$ as L. Let $(C_0 - \{s_i\} \cup \{q\})$ be the initial configuration of B_i. Algorithm B_i operates as follows: if it already has a server at the point of an incoming request r_i it does nothing. Otherwise it moves the server currently occupying the point r_{i-1} to serve r_i. Let now $C_i(j)$ denote the configuration of algorithm B_i immediately after it has served r_j. A simple induction on j reveals that for every j, $0 \leq j \leq m$, and every $i \neq l$,

$$C_i(j) \neq C_l(j).$$

This means that every request r_j forces exactly one of the algorithms B_i to move a server in order to service it. The others already have a server at r_j. As a result,

$$\sum_{i=1}^{k} \mathrm{cost}_{B_i}(\boldsymbol{\rho}) = \sum_{j=1}^{m-1} d(r_j, r_{j+1}) \leq \mathrm{cost}_L(\boldsymbol{\rho}).$$

which implies that there exists at least one algorithm B_i such that $\mathrm{cost}_{B_i}(\boldsymbol{\rho}) \leq \frac{1}{k}\mathrm{cost}_L(\boldsymbol{\rho})$.

Taking into account the fact that the initial configurations of L and B_i are not the same, the triangle inequality implies that

$$\mathrm{cost}_{OPT}(\boldsymbol{\rho}) \leq \mathrm{cost}_{B_i}(\boldsymbol{\rho}) + \max_{s_i \in C_0} d(s_i, q) \Longrightarrow \mathrm{cost}_L(\boldsymbol{\rho}) \geq k\mathrm{cost}_{OPT}(\boldsymbol{\rho}) - k \max_{s_i \in C_0} d(s_i, q)$$

where $\mathrm{cost}_{OPT}(\boldsymbol{\rho})$ is the optimal cost of serving the request sequence $\boldsymbol{\rho}$ starting at C_0. From that, the lemma follows. $\square$

Karloff et al. ([14]) address the question of lower bounds for randomized on-line k-server algorithms in unrestricted metric spaces. Through a lengthy argument they were able to prove that there exists an $\Omega(\log\log k)$ lower bound on the competitive ratio of every randomized on-line algorithm (against oblivious adversaries) for every symmetric space with at least $(k+1)$ points and that this lower bound can be strengthened to $\Omega(\log k)$ if the space is large enough to include a $(k+1)$ size subspace that is almost *superincreasing*. Roughly speaking, a space $(Q = \{q_0, \ldots, q_n\}, d)$ is superincreasing if $d(q_i, q_{i+1})$ is much larger than $d(q_{i-1}, q_i)$. First they prove that there can be no randomized on-line k-server algorithm with a sublogarithmic competitive ratio in a superincreasing space with at least $(k+1)$ points. The next step is to prove that a large enough space will contain either a subspace with $(k+1)$ points that is almost superincreasing or a subspace (of the same size) that is almost uniform. In the later case, the $\Omega(logk)$ lower bound follows from the following result, proved by Fiat et al. ([11]), which addresses the performance of randomized algorithms in unifrom spaces.

Lemma 8.2 *No randomized on-line k-server algorithm for uniform space can be better than H_k-competitive against an oblivious adversary, where $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$ is the k-th harmonic number.*

Proof: Let L be any randomized on-line k-server algorithm for the uniform space (Q, d) and consider any starting configuration $C_0 = \{s_1, \ldots, s_k\}$ consisting of k distinct points. For an arbitrary point $q \in Q$ such that $q \notin C_0$ let $H = C_0 \cup \{q\}$. We will show how an oblivious adversary can construct a sequence $\boldsymbol{\rho}$ of requests at points of H so that the expected cost that L pays in serving $\boldsymbol{\rho}$ is at least H_k times the adversary cost for serving $\boldsymbol{\rho}$ (where both the adversary and the algorithm L start with their servers at C_0). The adversary generates the input sequence $\boldsymbol{\rho} = \boldsymbol{\rho}_1 \ldots \boldsymbol{\rho}_m$ in phases, where each phase $\boldsymbol{\rho}_i$ is a subsequence of requests such that

$$\text{cost}_{ADV}(\boldsymbol{\rho}_i) = 1 \qquad \text{and} \qquad E[\text{cost}_L(\boldsymbol{\rho}_i)] \geq H_k.$$

Since the adversary is oblivious, it is not allowed to see how the algorithm L will respond to a request before generating the new one. It does, however, know the code of L. This means that it can maintain an array $P[\ ]$ of probabilities where, for each $x \in H$

$$P[x] = \Pr\{\text{x is not covered by an algorithm server after the most recent request}\}.$$

Initially, $P[s_i] = 0$ $(\forall s_i \in C_0)$ and $P[q] = 1$. The entries of $P[\ ]$ are updated after the generation of every new request. The request sequence $\boldsymbol{\rho}$ is generated based on the value of P (observe that a request placed at a point r will make the algorithm pay an expected cost of $P[r]$ for serving r).

The adversary maintains a dynamic set of marks over the points of H. Initially, there are k marks, one on every point of C_0. When a new request r is generated, a mark is placed on r, if none is already there. If, however, after placing that mark all points in H are marked then all marks, except the one placed on r, are removed. A phase $\boldsymbol{\rho}_i$ ends as soon as there are k marked points in H. Every phase is broken down to k subphases. Each subphase consists of a number (possibly zero) of requests to already marked points followed by a request to an unmarked point. Let M always be the set of marked points and set $t = |H - M|$. At the begining of a subphase the set $H - M$ is checked for a point r such that $P[r] \geq 1/t$. If such a point exists, then a request is placed at r (causing L to pay an expected cost of $1/t$ in order to serve it) and the subphase ends. Otherwise, there must be at least one point $x \in M$ such that

$P[x] > 0$. Let $\varepsilon = P[x]$ for such a x and use U to always denote the current value of $\sum_{y \in M} P[y]$. The subphase then starts by generating requests to already marked points according to the following loop:

> While $U > \varepsilon$ and while the total expected cost of all the requests in this subphase so far does not exceed $1/t$, request a point $r \in M$ such that $P[r] = \max_{x \in M}\{P[x]\}$.

If the loop ends with the expected cost of the subphase exceeding $1/t$ then the subphase concludes with a request to an arbitrary unmarked point. Otherwise, the last request is for the unmarked point y with the highest $P[y]$ value. In either case it can be verified that the cost of L for the subphase is at least $1/t$.

The proof is completed by observing that in the duration of a phase t takes (in that order) the values $1, k, (k-1), \ldots, 2$ and summing over all subphases we can conclude that the cost of L for serving any phase $\boldsymbol{\rho}_i$ is at least H_k. Furthermore, the adversary can serve every phase at a unit cost. To see this, let r, r' be the first requests of phases $\boldsymbol{\rho}_i$ and $\boldsymbol{\rho}_{i-1}$ respectively. The definition of a phase implies that $r \neq r'$. If when r' arrives the adversary serves it with the server that it has at point r, then it can be verified that all subsequent requests of $\boldsymbol{\rho}_{i-1}$ fall on points already covered by adversary servers. $\square$

In the spirit of Lemma 6.2, the above result also implies that if (Q, d) is a metric space such that

$$\frac{\max_{x,y \in Q} d(x,y)}{\min_{x,y \in Q} d(x,y)} \leq b$$

for some $b > 0$ then no randomized on-line algorithm for the k-server problem in that space can be better than $\frac{H_k}{b}$-competitive against an oblivious adversary. Furthermore, combining Lemmas 8.2, 2.1 and 2.2 we can conclude that no randomized on-line algorithm for the caching problem can be better than k/H_k-competitive against an adaptive on-line adversary (or else there would be a better than k-competitive deterministic on-line caching algorithm).

Regarding the performance of memoryless randomized algorithms for the uniform space, Raghavan and Snir ([23]) show the following:

Lemma 8.3 *No memoryless randomized on-line algorithm for the caching problem can be better than k-competitive against an oblivious adversary.*

For the weighted cache problem, Chrobak et al. ([4]) prove that:

Lemma 8.4 *There exists no memoryless deterministic on-line algorithm that is c-competitive for every instance of the k-server weighted cache problem, no matter what the value of c is.*

Proof: Let L be such an algorithm that claims to be c-competitive ($c > 0$), for every instance of the weighted cache problem. Consider such an instance defined by the set of points $Q = \{q_1, \ldots, q_{k+1}\}$ together with the assignment of weights $w(q_i) = V^i$, for some positive number V. Let B be an adversary, starting in the same initial configuration as L, which generates the input request sequence in the following way. It starts by always placing a request to a point that is uncovered by the algorithm servers. It will keep on doing so until the algorithm goes through some configuration C for the second time. Since there are k servers and $(k+1)$ points this will not take more than $(k+1)^k$ requests to happen. At that poin the adversary modifies its strategy: from then on it simply repeats the sequence of requests in the cycle between the first and the second appearence of C. Call this sequence of requests $\boldsymbol{\rho}$. The memorylessness of L guarantess that the algorithm will always go through exactly the same configurations when faced with $\boldsymbol{\rho}$. Let now q_i be the highest indexed point in $\boldsymbol{\rho}$. It follows that for each repetition of the request sequence $\boldsymbol{\rho}$ the cost of L is at least V^i. The adversary on the other hand, always keeps one of its servers at q_i and uses any one of its remaining servers for handling the other requests. As a result, the adversary cost for serving $\boldsymbol{\rho}$ is no more than $(k+1)^k V^{i-1}$. Over a very large number of repetitions of $\boldsymbol{\rho}$ the costs incurred before reaching the cycle can be ignored. Thus, in the limit, the ratio of the algorithm and adversary cost is at least $V^i/((k+1)^k V_{i-1}) = V/(k+1)^k$. A suitably large choice of V makes this ratio exceed c, contradicting the claim that L is c-competitive. $\Box$

A similar argument can be used to prove the following:

Lemma 8.5 *There exists no memoryless deterministic on-line algorithm that is c-competitive for every instance of the on-line k-server problem, no matter what the value of c is.*

Chrobak et al. ([4]) show how to construct asymmetric metric spaces for which no deterministic on-line k-server algorithm can be c-competitive, for any value of c. This proves that, for the general case of the on-line k-server problem in asymmetric spaces, there exist no competitive deterministic algorithm.

9 Conclusion

In the past few sections we tried to exhibit some of the work performed during the last few years on the on-line k-server problem. The results, both positive and negative, that have been presented show the definite progress achieved in understanding and analyzing the problem. Despite this progress, though, there are quite a few tantalizing questions that remain unanswered.

The most eminent among them is the k-server conjecture: is it true that for every instance of the k-server problem over a symmetric metric space there exists a k-competitive deterministic algorithm? All the evidence seem to indicate that the answer is yes but a concrete proof still eludes us. The conjecture has been settled in the positive only for restricted versions of the problem: it is known to be true for the 2-server problem over any symmetric space ([18]) and for the k-server problem when the underlying space has $(k+1)$ points ([18]) and $(k+2)$

points ([16]); it has also been shown to hold for particular metric spaces (uniform spaces, trees). For the general case, though, the best result known to date is the $(2k-1)$ competitive ratio achieved by the work function algorithm. In fact, this algorithm is believed to really be k-competitive although no conclusive proof has been found. Chrobak and Larmore ([6]) report that computer experiments over thousands of small metric spaces have revealed no counterexample forcing a larger than k competitive ratio on the algorithm.

A similar question arises when considering randomized algorithms. Are there linearly competitive randomized algorithms? In fact, the apparent superiority of randomized versus deterministic algorithms, at least when playing against oblivious adversaries (emphatically displayed in the discussion of the k-server problem over uniform spaces), gives rise to an even more interesting possibility: are there randomized on-line algorithms that achieve a sublinear competitive ratio for the general case of the k-server problem? Of particular interest is the case of memoryless randomized algorithms. As the example of HARMONIC shows, such algorithms can be competitive despite their memorylessness property, something that is not true for their deterministic counterparts. What is the best competitive ratio for a memoryless, randomized k-server algorithm? Is it possible that it is linear in k? Since memoryless algorithms are usually easy to implement and fast, there is also an important practical side in answering these questions (especially in the face of the very bad execution time of the work function algorithm).

The only randomized and memoryless algorithm that has been shown to be competitive for the general case of the k-server problem is the HARMONIC algorithm. Although HARMONIC is, provably, not linearly competitive (neither against oblivious nor adaptive on-line adversaries) there still exists a large gap between the exponential competitive ratio that it has been shown to achieve and its worst known behaviour ($\frac{k(k+1)}{2}$-competitive against lazy adversaries). Finding the exact competitive ratio of HARMONIC remains an open problem.

References

[1] S. Ben-David, A. Borodin, R. Karp, G. Tárdos, and A. Wigderson. On the power of randomization in online algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 379–386, 1990.

[2] P. Berman, H. Karloff, and G. Tárdos. A competitive 3-server algorithm. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 280–290, 1990.

[3] A. Borodin, N. Linial, and M. Saks. An optimal on-line algorithm for metrical task systems. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 373–382, 1987.

[4] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New results on server problems. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1990.

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

[6] M. Chrobak and L. Larmore. The server problem and on-line games. In *On-Line Algorithms: Proceedings of a DIMACS Workshop*, volume 7 of *DIMACS Series in Discrete Mathematics*, pages 11–64. 1992.

[7] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. Random walks on weighted graphs, and applications to on-line algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 369–378, 1990.

[8] X. Deng and S. Mahajan. Server problems and resistive spaces. *Information Processing Letters*, 37:193–196, 1991.

[9] P. Doyle and J. Snell. *Random Walks and Electric Networks*. Number 22 in The Carus Mathematical Monographs. The Mathematical Association of America, 1984.

[10] D. Du and F. Hwang. Competitive group testing. In *On-Line Algorithms: Proceedings of a DIMACS Workshop*, volume 7 of *DIMACS Series in Discrete Mathematics*, pages 125–134. 1992.

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

[12] A. Fiat, Y. Rabani, and Y. Ravid. Competitive k-server algorithms. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 454–463, 1990.

[13] E. Grove. The harmonic k-server algorithm is competitive. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 260–266, 1991.

[14] H. Karloff, Y. Rabani, and Y. Ravid. Lower bounds for randomized k-server and motion planning algorithms. 1991.

[15] E. Koutsoupias and C. Papadimitriou. Beyond competitive analysis. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 394–400, 1994.

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

[17] L. Larmore M. Chrobak. Harmonic is 3-competitive for two servers. 1990.

[18] M. Manasse, L. McGeogh, and D. Sleator. Competitive algorithms for server problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 322–333, 1988.

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

[20] S. Phillips and P. Tetali. Hitting costs via electrical resistances and the harmonic algorithm for k-servers. 1993.

[21] P. Raghavan. Lecture notes on randomized algorithms. Research report, IBM Research Division, T.J. Watson Research Center, 1990.

[22] P. Raghavan. A statistical adversary for on-line problems. In *On-Line Algorithms: Proceedings of a DIMACS Workshop*, volume 7 of *DIMACS Series in Discrete Mathematics*, pages 79–83. 1992.

[23] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. In *16th International Colloqium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 687–703. Springer-Verlag, 1989.

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