18.415/6.854 Advanced Algorithms	September 1994

On-Line Algorithms

Lecturer: Michel X. Goemans

1 Introduction

The first topic we will cover in this course is *on-line algorithms*. The concept of on-line algorithms has been around for quite some time (it was for example present in the seventies in the bin packing literature, see [13, 20]), but it has been a fairly hot research area since the publication of the landmark paper by Sleator and Tarjan [19] in 1985.

Typically when we solve problems and design algorithms we assume that we know all the data *a priori*. However in many practical situations this may not be true and rather than have the input in advance it may be presented to us as we proceed. We give below some examples to motivate the topic of on-line algorithms.

1.1 Investment Problem

Consider an investor with a given sum of money which he wishes to invest so as to maximize his gain at the end of a specified period of time. He has a variety of options: he may keep funds in a money market account, he may buy certificates of deposit or he may invest in the stock market. He can keep a mixed portfolio and he can reallocate his assets during the course of the investment period. Now in the offline case he has full information about the behaviour of the various financial and capital markets and so can compute an optimal strategy to maximize his profit. An on-line algorithm, however, is a strategy which at each point in time decides what portfolio to maintain based only on past information and with no knowledge whatsoever about the future. The profit generated is a determinant of the quality of the on-line strategy. We will see later how exactly to quantify this quality by introducing the notion of competitive analysis.

1.2 Ski Rental Problem

Consider the following scenario: you are a skier and, each day, need to either rent skis for \$1 or buy a pair of skis for \$$T$ which will last for the rest of the season. Unfortunately, you do not know when the ski season will end. If you knew a priori the length of the season (this is the offline case) say L then it is obvious that you should rent if $L < T$ and buy if $L \geq T$. On the other hand, an on-line strategy for this problem would first fix an integer k, and then rent for k days and buy on day $k+1$ (if it has the opportunity).

At this point we must pause and decide how to evaluate the performance of on-line algorithms? One way to analyze such algorithms is to assume that the input is given according to a certain probability distribution, and compute the expected behavior of the algorithm based on this distribution. However we wish to avoid making assumptions about input distributions. The analytic machinery we present will not demand that the inputs come from some known distribution, but instead will compare the performance of the on-line algorithm with that of the best offline algorithm. This notion of comparison is called *competitive analysis* and was introduced by Sleator and Tarjan [19].

Definition 1 *An on-line algorithm* A *is α-competitive if for all input sequences σ, $C_{\mathrm{A}}(\sigma) \leq \alpha C_{\mathrm{MIN}}(\sigma)$, where $C_{\mathrm{A}}(\sigma)$ is the cost of the on-line strategy* A *for σ and $C_{\mathrm{MIN}}(\sigma)$ is the cost of the optimal offline algorithm for σ.*

Let us now reconsider the ski rental problem in the light of the above definition. One strategy is to buy on the first day. This strategy is T-competitive and the worst input sequence is obtained when $L = 1$. Another on-line strategy is to rent for the first $T-1$ days and then buy. If $L < T$ the cost of the on-line and optimal offline strategy is the same. If $L \geq T$ the cost of the on-line strategy is $2T-1$ and the cost of the optimal offline strategy is T. Hence this algorithm is $(2-1/T)$-competitive and it is fairly easy to show that it is the optimal on-line algorithm.

In many cases, the definition of competitiveness is slightly relaxed by imposing that $C_{\mathrm{A}}(\sigma) \leq \alpha C_{\mathrm{MIN}}(\sigma) + c$ for some constant c independent of σ.

2 Paging Problem

This is a problem which arises in system software. We have a two level memory divided into pages of fixed size. There are k pages in fast memory while the remaining pages are in slow memory. The input sequence σ in this problem are requests to pages $< \sigma_1, \sigma_2, \sigma_3 \cdots >$. If the request for a page σ_i is in fast memory then we need not do anything, otherwise we have a page fault and are faced with the problem of deciding which page in the fast memory to swap with σ_i. Our cost in this case is the number of page faults. An on-line algorithm for this problem decides in the case of a page fault which page to remove from fast memory without any information of the subsequent page requests. For example, here are some possible on-line strategies:

LIFO (Last-In-First-Out): remove the page most recently placed in fast memory.

FIFO (First-In-First-Out): remove the page that has been in memory longest.

LRU (Least-Recently-Used): replace the page whose most recent access is earliest.

LFU (Least-Frequently-Used): replace the page that has been accessed the least.

FWF (Flush-When-Full): remove any unmarked page and mark the requested page. Whenever all pages in fast memory are marked, unmark them all.

Let us consider the LIFO strategy for the paging problem. Consider an input sequence of two pages p and q repeatedly requested one after the other. It is easy to see that LIFO is not α-competitive for any α. We say that LIFO is *not competitive.* A similar comment can be made about LFU.

In the theorem to follow we will show that there exist k-competitive strategies (where k is the size of the fast memory). Later, we shall show that this is the best possible and we shall also characterize MIN - the optimal offline algorithm.

Theorem 1 LRU *is k-competitive.*

Proof:

Note that by definition of our problem, both LRU and MIN have the same initial configuration in fast memory (i.e. the same k pages in fast memory.) We must show that for any k initial pages in fast memory, and for all σ, $C_{\mathrm{LRU}}(\sigma) \leq k \cdot C_{\mathrm{MIN}}(\sigma)$.

Let us examine actions of LRU for any particular fixed σ. Let us divide the input sequence into phases:

$$\sigma = \sigma_1, \ldots, \underbrace{\sigma_i}_{\text{first page fault}}, \underbrace{[\sigma_{(i+1)}, \ldots, \sigma_j]}_{\text{phase 1}}, \underbrace{[\sigma_{(j+1)}, \ldots, \sigma_l]}_{\text{phase 2}}, \ldots$$

where each phase contains exactly k page faults, ending with a page fault. For example, the first phase ends with σ_j where

$$j = min\{t \ : \ \text{LRU has } k \text{ page faults in } \sigma_{(i+1)}, \ldots, \sigma_t\}$$

Let us consider a cost of a single phase for both LRU and MIN. By definition, LRU makes k faults. We argue that MIN must make at least one page fault on each phase. Let us consider two different cases.

Case 1: during the same phase, LRU faults twice on some page p. Thus, the phase looks likes this:

$$\ldots, [\sigma_{(i+1)}, \ldots, \overbrace{\underbrace{\sigma_{p_1} = p}_{\text{first page fault } p}, \ldots, \underbrace{\sigma_{p_2} = p}_{\text{the second page fault of } p},}^{\text{all } k \text{ pages in fast memory have been requested before second request to } p} \ldots, \sigma_j], \ldots$$

Notice that after the first page fault to page p, p is brought into fast memory and is later replaced (by some other page) only if p becomes the least recently used page in fast memory (by definition of LRU.) Hence, if there is another page fault to p, it means that all other k pages that are resident in fast memory just before σ_{p_2} came after the first page fault to p but before the second request to p. Thus, in this case, $k+1$ different pages have been requested in a phase. This, however, forces MIN to make at least one page fault in every phase , i.e. $C_{\mathrm{MIN}} \geq 1$ for every phase.

Case 2: LRU faults on k different pages. Here, we consider two sub-cases, depending on the last page-fault (say on p) which occurred before the beginning of current phase:

$$\underbrace{\sigma_i = p}_{\text{last page fault before current phase}}, \underbrace{[\sigma_{(i+1)}, \dots, \sigma_j]}_{\text{current phase}},$$

Case 2a: In a current phase, there is a page-fault to page p.

$$\underbrace{\sigma_i = p}_{\text{last page fault before current phase}}, \underbrace{[\sigma_{(i+1)}, \dots, p, \dots, \sigma_j]}_{\text{current phase}},$$

This case is very similar to case 1. In particular, before the second page fault to p occurs, there must have been k requests to pages other then p, to force p to be swapped out. Thus, during the current phase there are a total of $k+1$ different page requests. Thus $C_{\text{MIN}}(phase) \geq 1$.

Case 2b: In a current phase, there is no page-fault to page p.

$$\sigma_i = p, \quad \underbrace{[\sigma_{(i+1)}, \dots, \sigma_j]}_{\text{current phase with no page fault on } p}$$

Let us consider MIN's behavior. Before the phase starts it must have p in its fast memory, and other $k-1$ locations which we don't care about. Notice, however, that during current phase, k different requests arrive, none of them are p. Thus, to accommodate all of them, MIN must get rid of p, hence $C_{\text{MIN}}(phase) \geq 1$ as well.

Thus, for every phase, MIN must pay at least one page fault, wile LRU pays k faults. However, what about the very first page fault, before the beginning of the first phase? Observe that both LRU and MIN start in the same configuration. Hence, the first time there is a request to page which in not in memory, it is in fact not in memory for both LRU and MIN. Hence they both must do a page fault. □

Remark 1 *Notice that essentially the same argument goes for* FIFO *in order to show that* FIFO *is also k-competitive.*

The above theorem was proved by Sleator and Tarjan [19]. Notice that in the above proof we have not used explicitly what is the optimal offline strategy MIN. We have only related the cost of the on-line algorithm under consideration to a *lower bound* on the cost of any on-line algorithm. This observation might seem elementary but comparing to a lower bound is a very typical technique in the analysis of algorithms (not just on-line algorithms but also, as we shall see later in the class, approximation algorithms).

The moral of the above proof-technique could be summarized as follows: we can view competitiveness as a game between A's strategy and an adversary who comes up with very bad inputs to A, knowing ahead of time A's strategy. That is, we can view it as a "game" between a player A, and an adversary. The adversary make the first move by selecting σ_1 and then the player has to process this request. Then the adversary selects (as his second move) σ_2 and the player processes σ_2 and so on. Of course, the adversary knowing A's strategy in advance can simulate the game and therefore can find (ahead of time) and play the most nasty sequence for A.

3 An Optimal Off-Line Algorithm for Paging

In this section, we determine an optimal off-line paging algorithm. We consider an off-line algorithm which we call *longest forward distance* or LFD. When LFD must service a page fault it removes the page, among the pages currently in memory, whose next request comes last.

Theorem 2 (Belady [1]) LFD *is an optimal off-line algorithm for paging.*

Proof:

By contradiction: assume that there is an algorithm MIN superior to LFD. Let $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ be a sequence so that $C_{\mathrm{LFD}}(\sigma) > C_{\mathrm{MIN}}(\sigma)$. At some point in the sequence the two algorithms diverge. Let σ_i be the page request which initiates this divergence. Before processing σ_i the two algorithms have the same pages in memory so that σ_i must be a fault for both. Since they diverge at σ_i, they discard different pages to service σ_i. Let q be the page that LFD discards and p the page that MIN discards. Let t be the first time after i that MIN discards q. We will alter MIN's behavior between σ_i and σ_t to create a new algorithm MIN*. MIN* will service σ_i the same way that LFD does (by removal of q) and will satisfy $C_{\mathrm{MIN}*}(\sigma) \leq C_{\mathrm{MIN}}(\sigma)$.

By the definition of LFD, the next request to p, say σ_a, comes before the next request to q, say σ_b, so that

$$\sigma = \sigma_1 \ldots \sigma_i \ldots \sigma_a = p \ldots \sigma_b = q \ldots \sigma_n$$

MIN* services σ_i by discarding q (as does LFD). After processing σ_i, then, MIN* and MIN share exactly $k-1$ pages of memory. As we describe the behavior of MIN* we will argue that, until MIN and MIN* converge, they share exactly $k-1$ pages of memory. MIN* behaves as follows:

CASE **1:** $i < t < b$. We describe how MIN* services a page request σ_l for $i < l < t$. By induction on l, we will show that MIN and MIN* share exactly $k-1$ pages of memory. This implies that there is a unique page e which MIN* has in fast memory but MIN does not. Initially, i.e. for $l = i+1$, this page e is p.

$\sigma_l \neq e$: When $\sigma_l \neq e$, MIN faults exactly when MIN* faults. When there is a fault, MIN* discards the same page that MIN does (recall that MIN does not discard q until time t so that MIN* can always do this). If MIN and MIN* shared $k-1$ pages before the request of σ_l then they share $k-1$ pages afterwards.

$\sigma_l = e$: When $\sigma_l = e$, MIN faults but MIN* does not. In this case, MIN replaces some page with e. Again, if MIN and MIN* shared $k-1$ pages before the request of σ_l then they share $k-1$ pages afterwards.

Finally, to service σ_t, which is a fault for both, MIN* discards e. Since MIN (by definition) discards q, MIN and MIN* again have converged. From the above descriptions, it is clear that $C_{\text{MIN}*}(\sigma) \le C_{\text{MIN}}(\sigma)$.

CASE **2:** $t \ge b$. In this case, MIN* follows the same strategy as above for σ_l where $i < l < b$. Notice that the same analysis is applicable and, upon arrival at $\sigma_b = q$, MIN and MIN* have exactly $k-1$ pages in common. Further notice that, since p is requested before q, the events described in the ($\sigma_l = e$) portion of the above discussion have taken place at least once so that $C_{\text{MIN}*}(\sigma_1 \ldots \sigma_{b-1}) < C_{\text{MIN}}(\sigma_1 \ldots \sigma_{b-1})$. By definition, MIN does not fault on $\sigma_b = q$. MIN* does and replaces e (the unique page which it has in memory that MIN does not) with q. MIN* and MIN then have the same pages in memory after servicing σ_b. MIN* then behaves as MIN does for the remainder of σ. Since $C_{\text{MIN}*}(\sigma_1 \ldots \sigma_{b-1}) < C_{\text{MIN}}(\sigma_1 \ldots \sigma_{b-1})$, $C_{\text{MIN}*}(\sigma) \le C_{\text{MIN}}(\sigma)$, as desired.

We conclude that $C_{\text{LFD}}(\sigma_1 \ldots \sigma_i) \le C_{\text{MIN}*}(\sigma_1 \ldots \sigma_i)$, an optimal strategy, and by induction that $C_{\text{LFD}}(\sigma) \le C_{\text{MIN}}(\sigma)$. □

4 A Lower Bound on the (Deterministic) Competitive Ratio for the Paging Problem

We show a lower bound of k for the competitive ratio of any deterministic on-line algorithm for the paging problem. We conclude that LRU is an optimal deterministic on-line algorithm for this problem.

Theorem 3 *For all on-line algorithms* A *there is a sequence of requests* $\sigma^{\text{A}} = \sigma_1^{\text{A}} \ldots \sigma_n^{\text{A}}$ *so that* $C_{\text{A}}(\sigma^{\text{A}}) \ge k \cdot C_{\text{LFD}}(\sigma^{\text{A}})$. *This sequence* σ^{A} *can in fact be chosen from a universe of only* $k+1$ *pages.*

Proof:

Given A, let σ_i^{A} be the (unique) page excluded from the k-page memory of A after servicing $\sigma_1^A \ldots \sigma_{i-1}^{\text{A}}$. With this sequence A faults on every request. Then, assuming that the memory of A begins in some full initial state, we have that for all prefixes σ of σ^A, $C_{\text{A}}(\sigma) = |\sigma|$. For LFD, however, we have:

Lemma 4 *For all finite sequences σ chosen from a universe of $k+1$ pages* $C_{\mathrm{LFD}}(\sigma) \leq \frac{|\sigma|}{k}$.

Proof of Lemma:

Suppose that σ_i induces a page fault for LFD. We show that LFD incurs no page faults on any of $\sigma_{i+1}, \ldots, \sigma_{i+k-1}$. Let p be the page which LFD discards to service σ_i. Since there are exactly $k+1$ pages, the next fault of LFD must be made on a request to p. Notice that every other page in memory (before the service of σ_i) must be requested *before* p is next requested (p is, by definition, requested last among these pages). Hence at least $k-1$ requests separate σ_i from the next fault, as desired. □

Let σ be a prefix of σ^{A} of length kl. Then $C_{\mathrm{A}}(\sigma) = kl \geq k \cdot C_{\mathrm{LFD}}(\sigma)$ so that A is, at best, k-competitive. □

Remark 2 FIFO *is also a k-competitive on-line algorithm for the paging problem and so is also optimal.*

5 Randomized On-Line Algorithms

The on-line algorithms we have discussed up to this point have been deterministic. One shortcoming of such algorithms is that an adversary can always exactly determine the behavior of the algorithm for an input σ. This leads to diabolical inputs like that crafted in the previous theorem. This motivates the introduction of the class of randomized on-line algorithms which will have better behavior in this respect. Informally, a randomized on-line algorithm is simply an on-line algorithm that has access to a random coin. Formally,

Definition 2 *A* randomized on-line algorithm A *is a probability distribution* $\{\mathrm{A}_x\}$ *on a space of deterministic on-line algorithms.*

Notice that when we talk about randomized on-line algorithms, we must decide what information the adversary is allowed to have. If we wish to say that the adversary does not see any coin-flips of the algorithm (or, equivalently, that the adversary must select his "nasty" sequence in advance, when he has no knowledge of the actual pages removed from memory) then we define

Definition 3 *An oblivious adversary knows the distribution on the deterministic on-line algorithms induced by* A*, but has no access to its coin-tosses.*

The above definition also implies that the adversary can not inspect which pages A holds in his fast memory. Equivalently, this means that the oblivious adversary must construct his nasty sequence σ before the game between him and randomized A starts. Thus, notice that against an oblivious adversary, randomization is useful in order to hide the status of the on-line algorithm. The notion of competitiveness is now defined as follows:

Definition 4 *A randomized on-line algorithm* A *distributed over deterministic on-line algorithms* $\{\mathrm{A}_x\}$ *has a* competitive ratio *of* α *against any oblivious adversary if*

$$\exists c, \forall\sigma, \operatorname*{Exp}_x[C_{\mathrm{A}_x}(\sigma)] \leq \alpha C_{\mathrm{MIN}}(\sigma) + c.$$

The additional constant c is introduced to account for possible differences in the initial configuration of the on-line algorithm and the adversary.

We introduce two randomized on-line paging algorithms:

RANDOM: RANDOM services page faults by discarding a random page, i.e. a page selected uniformly from the pages in fast memory.

MARKING: Initially, all pages are *marked.* When a page p is requested,

1. If p is not in memory then
   - If all pages in memory are marked then unmark all pages
   - Swap p with a uniformly selected unmarked page in memory.
2. Mark page p in memory.

We will first focus on MARKING. We will show that

- By using *randomization*, MARKING achieves a competitive ratio of $2H_k$ against an oblivious adversary, where $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$ is the k-th harmonic number.
- MARKING's competitive ratio is nearly optimal in that no randomized algorithm has competitive ratio better than H_k. In proving this, we will introduce a general method of proving lower bounds for randomized competitive algorithms against an oblivious adversary.

Both results are due to Fiat et al. [9].

6 Analysis of the MARKING Algorithm

Theorem 5 MARKING *is a $2H_k$-competitive paging algorithm against any oblivious adversary, where $H_k = \sum_{i=1}^{k} \frac{1}{i}$ is the kth harmonic number.*

In other words, we need to show that

$$\forall\sigma, \mathrm{Exp}[C_{\mathrm{M}}(\sigma)] \leq 2H_k C_{\mathrm{MIN}}(\sigma),$$

where M for conciseness denotes MARKING. This implicitly assumes that M and MIN have initially the same pages in fast memory (otherwise, we would have an additional constant c). To prove this competitiveness, we need to prove an upper bound on the LHS and a lower bound on the RHS, and show that the inequality still holds.

We begin by fixing the input sequence σ and by then dividing it into phases as follows.

- The first phase begins on the first page fault.
- The $i+1$-st phase starts on the request following the last request of phase i.
- If phase p starts on σ_{i_p} then it ends on $\sigma_{i_{p+1}-1}$ where $|\{\sigma_{i_p}, \ldots, \sigma_{i_{p+1}}\}| = k+1$ but $|\{\sigma_{i_p}, \ldots, \sigma_{i_{p+1}-1}\}| = k$.

Thus, in a phase, exactly k distinct pages are requested.

Notice that the initial portion before the first phase costs nothing. Hence the cost of M on σ is the sum of the costs of M on all phases of σ.

Before we go any further let us understand the dynamics of a phase by the following remarks. Let us denote by S_i the set of pages that were in memory just before phase i begins.

Remark 1: Once a page has been accessed in a phase it is marked and hence will remain in memory till the end of the phase. Thus we may pay a price of 1 for a page q only if this is is the first time q is accessed during this phase and if q is not in memory.

Remark 2: We claim that at the end (or the beginning) of every phase the k pages in fast memory are all marked and correspond to the k distinct pages requested during that phase. This can easily be shown by induction. At the beginning of the first phase all pages in fast memory are marked (since no page fault has yet occurred). By induction, one can thus assume that the first request in a phase will cause a page fault (this is true for phase 1 and, for later phases, this follows by induction from the claim applied to the previous phase). Then, during the phase, the k distinct pages requested will be marked and will remain in fast memory until the end of the phase, proving the inductive statement.

Remark 3: Obviously, the definition of a phase does not depend on the coin-tosses of M but only on the input sequence. In other words, the same input sequence will always be divided in exactly the same way into phases regardless of the coin-tosses of M. The coin-tosses only affect the dynamics of M's behavior within a phase. Also (see remark 2), S_i does not depend on the coin-tosses – S_i is simply the k distinct requested pages in phase $i-1$.

By these remarks we can focus on the k distinct pages accessed in phase i; for each such page, we only concentrate on the first time this page is accessed since only such first-time requests can cost us anything. We divide such first time requests into two categories.

- **Clean Requests**: These are requests to pages that did not belong to S_i. We have to pay a price of 1 for each such request regardless of the coin-tosses.
- **Stale Requests**: These are requests to pages that belonged to set S_i at the beginning of the phase. Unfortunately, such a page may have been ejected to make room for a clean page. If it is still around, we pay 0 or else we pay 1. Thus our first major task is to find the expected cost of a stale request.

Since a phase consists of requests to exactly k distinct pages, if there are l clean requests then there must be $k - l$ stale requests.

6.1 The expected cost of a stale request within a phase

Consider a typical stale request σ_j to page p within phase i. Assume there have been s stale requests and c clean requests so far in the phase. We wish to find the expected cost of this request in terms of c and s. To do so, we need the probability that page p is still in memory after s stale and c clean requests. Notice that since this is the first time we are accessing p, p is unmarked.

Now what do we know? We know that the pages requested by the s stale requests are in S_i, by definition of a stale request. Furthermore, since requested pages are marked, the stale pages will not be removed until the end of the phase, therefore they are in fast memory at time $j-1$. This means that both fast memory at time $j-1$ and S_i contain all of these s stale pages. Therefore, the c clean requests now occupy c uniformly distributed slots from the remaining $k-s$ slots. Now we can easily compute the expected cost of a stale request. It is the probability that the requested page is one of the c pages which is no longer in fast memory due to the clean requests. There are $k-s$ candidates for these pages. Hence:

$$\mathrm{Exp}[C_{\mathrm{M}}(\sigma_j)] = \frac{c}{k-s}.$$

6.2 Bounding the total cost of M on a phase

We summarize the last section as follows:

Lemma 6 *The expected cost of the $s+1$-st stale request in a phase is equal to $\frac{c}{k-s}$ where c is the number of preceding clean requests in this phase.*

Suppose we know that there are l_i clean requests within phase i. Recall that l_i is defined purely in terms of the input sequence (see Remark 3) and does not depend on the coin-tosses.

Then there must be $k - l_i$ stale requests within this phase. Clearly the number of clean requests preceding any stale request is no more than l_i. Using this and the last lemma and summing over all $k - l_i$ stale requests the expected cost of all the stale requests on the phase is no more than D where

$$D = \frac{l_i}{k} + \frac{l_i}{k-1} + \ldots + \frac{l_i}{k-(k-l_i-1)}.$$

Also the cost of all all clean requests is exactly l_i since each clean request costs 1. Hence the total expected cost of M on phase i is no more than $D + l_i$. Now

$$D + l_i = l_i(1 + \frac{1}{l_i+1} + \frac{1}{l_i+2} + \ldots + \frac{1}{k}) \leq l_i H_k.$$

In summary:

Lemma 7 *The expected cost of* M *on phase* i *of* σ *is no more than* $l_i H_k$ *where* $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$ *is the* $k-th$ *Harmonic number.*

6.3 Bounding the total cost of any offline algorithm M on a phase

Consider any offline algorithm A. Let us simulate A and M on the same input sequence σ. As before we can divide σ into phases defined by the execution of M. Let us define a potential function Φ_i which is the number of pages in A's memory that are not in M's memory just before Phase i begins. Notice that this potential function is well-defined because the pages in M's memory at the start of Phase i is determined by σ and not by the coin-tosses of M. (see Remark 3 again!).

We know that M receives l_i clean requests in phase i. By the definition of clean requests these were not in M's memory at the start of Phase i. Hence at least $l_i - \Phi_i$ of these pages are also not in A's memory at the start of Phase i. Thus if we let $C_i(\mathrm{A})$ be the cost of A during phase i we have

Lemma 8 $C_i(A) \geq l_i - \Phi_i$.

Next, by definition A has Φ_{i+1} pages at the end of Phase i that are not in M's memory. Thus M has a set P_i of Φ_{i+1} pages at the end of Phase i that are not in A's memory. But each page in M's memory at the end of Phase i was accessed in Phase i. Thus all pages in P_i must have been in $A's$ memory at some time during Phase i but they have been ejected by the end of the phase. Each ejection costs 1 and thus $C_i(\mathrm{A})$ must be at least $|P_i| = \Phi_{i+1}$. Thus:

Lemma 9 $C_i(\mathrm{A}) \geq \Phi_{i+1}$.

We can easily combine Lemmas 8 and 9 to get the more useful bound of $C_i(\mathrm{A}) \geq \frac{1}{2}(l_i - \Phi_i + \Phi_{i+1})$. If we now amortize this bound over the first n phases, we see that

$$\begin{aligned} C(A) &\geq \frac{1}{2}(l_1 - \Phi_1 + \Phi_2 + l_2 - \Phi_2 + \Phi_3 \ldots + l_n - \Phi_n + \Phi_{n+1}) \\ &= \frac{1}{2}(\sum_{i=1}^{n} l_i - \Phi_1 + \Phi_{n+1}). \end{aligned}$$

Since $\Phi_{n+1} \geq 0$ and we assume that $\Phi_1 = 0$, we derive:

Lemma 10 $C(\mathrm{A}) \geq \frac{1}{2}\sum_{i=1}^{n} l_i$

6.4 Establishing the Competitive Factor of M

From Lemma 7 the total expected cost of M on input σ is at most $H_k \sum_i l_i$. Also from Lemma 10 the total cost of any offline algorithm A on input σ is at least $0.5 \sum_i l_i$. Thus M is $2H_k$ competitive.

7 Lower Bound for any Randomized Algorithm

7.1 A useful technique

How do we prove a lower bound on the competitive factor α (even if we allow an additional constant) of a randomized algorithm against an oblivious adversary? We do so by picking a distribution D of input sequences and evaluating the *expected cost* of the best on-line algorithm and the *expected cost* of MIN (i.e. the best offline algorithm) on distribution D. We now develop this method.

Any randomized algorithm can be considered as a distribution A over all possible deterministic on-line algorithms A_x. Let $C_{\mathrm{H}}(\sigma)$ be the cost of deterministic algorithm H on input σ. Thus when we say that A is α-competitive we mean that for all σ,

$$\operatorname*{Exp}_x[C_{\mathrm{A}_x}(\sigma)] \leq \alpha C_{\mathrm{MIN}}(\sigma) + c.$$

Let us use σ^j to denote the first j elements in sequence σ. Assume we have a distribution D over the input sequences σ. Fix a j. We can take expectation (over input sequences of length j using distribution D) over both sides of the last equation to get

$$\operatorname*{Exp}_y[\operatorname*{Exp}_x[C_{\mathrm{A}_x}(\sigma_y^j)]] \leq \alpha \operatorname*{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)] + c.$$

We can use Fubini's theorem to exchange the two expectation quantifiers on the LHS and derive:

$$\operatorname*{Exp}_x[\operatorname*{Exp}_y[C_{\mathrm{A}_x}(\sigma_y^j)]] \leq \alpha \operatorname*{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)] + c.$$

Let $m_j = Min_{\mathrm{H}}(\mathrm{Exp}_y[C_{\mathrm{H}}(\sigma_y^j)])$ where the minimum is taken over deterministic on-line algorithms H. In other words m_j is the expected cost of the best on-line deterministic algorithm on input sequences distributed according to D. Using this definition in the last equation we have:

$$m_j \leq \alpha \operatorname*{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)] + c$$

and thus

$$\frac{m_j}{\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]} \leq \alpha + \frac{c}{\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]}.$$

Suppose D is chosen so that $\lim_{j\to\infty} \mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]$ is ∞. Then:

$$\lim_{j\to\infty} \frac{m_j}{\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]} \leq \alpha.$$

What is this equation saying? It says that the competitive ratio of any randomized algorithm against an oblivious adversary is at least as big as the ratio of the expected cost of the best deterministic on-line algorithm to the expected cost of the best offline algorithm when evaluated using distribution D over "large enough" input sequences. Notice that we are free to choose D to maximize this ratio.

7.2 Applying the general method to the Paging problem

We will prove the following result using our general method.

Theorem 11 *For the paging problem, if* A *is a randomized on-line* α*-competitive algorithm against an oblivious adversary then* $\alpha \geq H_k$*, even if there are only a total of* $k+1$ *pages that can be requested.*

Proof:

In order to have the inequality $m_j \leq \mathrm{Exp}_x[\mathrm{Exp}_y[C_{\mathrm{A}_x}(\sigma_y^j)]]$ as tight as possible, we choose D such that any deterministic on-line algorithm performs equally badly (in expected cost). In our case this can be obtained by choosing σ_i uniformly among all $k+1$ pages and independent of all previous requests. Since the fast memory only contains k pages, any deterministic algorithm will pay an expected cost of $\frac{1}{k+1}$ for serving σ_i resulting in an expected cost of $\frac{j}{k+1}$. Thus $m_j = \frac{j}{k+1}$.

Our lower bound technique therefore implies that

$$\alpha \geq \lim_{j\to\infty} \frac{j}{(k+1)\,\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]}.$$

Using this, we can restate the claim of our theorem as

$$\lim_{j\to\infty} \frac{j}{\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]} = (k+1)H_k. \tag{1}$$

To prove this claim, we need to study the behavior of MIN. We divide σ into stochastic phases. Phase $i, i \geq 0$ consists of the requests indexed in $[X_i, X_i + 1, \ldots, X_{i+1} - 1]$ where $X_0 = 1$ and $X_{i+1} = min\{t : \{\sigma_{X_i}, \sigma_{X_i+1}, \ldots, \sigma_t\} = \{1, \ldots, k+1\}\}$.

Notice that the X_i are random variables since σ is distributed according to D (which in our case is the uniform distribution.) A phase contains requests to only k pages and hence if MIN has a page fault in some phase, the next page fault cannot occur before the next phase. Thus the number of page faults incurred by MIN on σ is at most the number of phases that occur up to time j. And hence the expected cost incurred by MIN is at most the expected number of phases that can occur up to time j. Hence

$$\mathop{\mathrm{Exp}}_{y}[C_{\mathrm{MIN}}(\sigma_y^j)] \leq 1 + \mathrm{Exp}[\max\{p : X_p \leq j\}].$$

To compute the LHS of (1) we use some results from the theory of stochastic processes. Since the random variables $\{Y_i\} = \{X_{i+1} - X_i : i \geq 0\}$ are independent and identically distributed, they form a so-called "renewal process" and by the elementary renewal theorem we have that

$$\begin{aligned}\lim_{j\to\infty}\frac{j}{1+\mathrm{Exp}[\max\{p:X_p\le j\}]} &= \lim_{j\to\infty}\frac{j}{\mathrm{Exp}[\max\{p:X_p\le j\}]}\\ &= \mathrm{Exp}[\text{length of phase}]\\ &= \mathrm{Exp}[X_1-1]=\mathrm{Exp}[X_1]-1.\end{aligned}$$

Intuitively, this says that the expected number of phases is asymptotically equal to the length of the sequence divided by the expected length of a phase. Removing the "expected" this makes perfect sense.

We have therefore shown that

$$\alpha \ge \frac{\mathrm{Exp}[X_1]-1}{k+1}.$$

We now need to compute $\mathrm{Exp}[X_1]$. By definition, $X_1 = min\{t : \{\sigma_1,\ldots,\sigma_t\} = \{1,\ldots,k+1\}\}$. Each σ_i is uniformly distributed among the $k+1$ pages and is independent of σ_j for any $j \neq i$. Computing $\mathrm{Exp}[X_1]$ under these circumstances is called the "coupon collector problem": Given a collection of $k+1$ distinct coupons (say baseball cards), $\mathrm{Exp}[X_1]$ represents the expected number of independently and uniformly selected coupons a collector has to acquire before obtaining the complete set of distinct coupons. In order to solve this problem, we introduce the random variables Z_i for $1 \le i \le k+1$, where $Z_i = min\{t : |\{\sigma_1,\ldots,\sigma_t\}| = i\}$ for $i = 1,\ldots,k+1$. Thus $Z_1 = 1$ and $Z_{k+1} = X_1$. Note that between Z_i and Z_{i+1}, the probability of getting a new page is $\frac{k+1-i}{k+1}$ at each step. So waiting for a new page between Z_i and Z_{i+1} is a Bernoulli process and the expected waiting time is $\frac{k+1}{k+1-i}$. Hence $\mathrm{Exp}[Z_{i+1}-Z_i] = \frac{k+1}{k+1-i}$. Thus

$$\begin{aligned}\mathrm{Exp}[X_1] &= \mathrm{Exp}[Z_{k+1}] = \Sigma_{i=1}^{k}(\mathrm{Exp}[Z_{i+1}]-\mathrm{Exp}[Z_i]) + \mathrm{Exp}[Z_1]\\ &= (\sum_{i=1}^{k}\mathrm{Exp}[Z_{i+1}-Z_i]) + 1\\ &= (k+1)(1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{k})+1\\ &= (k+1)H_k+1.\end{aligned}$$

So $\alpha \ge \frac{[(k+1)H_k+1]-1}{k+1} = H_k$. This completes the proof of Theorem 11. □

From Theorem 11, we see that Marking is optimal up to a factor of 2. An optimal (i.e. with competitive ration H_k) randomized on-line algorithm against any oblivious adversary has been obtained by McGeoch and Sleator [17].

8 Types of Adversaries

In this section, we discuss various types of adversaries for on-line algorithms and their relative power. The definitions and results in this and the following section are due to Ben-David, Borodin, Karp, Tardos, and Wigderson [2].

So far, we have analyzed the performance of a randomized on-line algorithm A against an *oblivious* adversary Q. Q knows the distribution over deterministic algorithms that A is using, but must generate a request sequence σ without knowledge of the results of A's coin tosses. We have analyzed A's performance in terms of a "game" between A and Q. Before the beginning of the game, Q selects a number n to be the number of requests $\sigma_1, \sigma_2, \ldots, \sigma_n$ which it will make. Then Q begins to make requests and A begins to process them on-line with "answers" $a_1, a_2, \ldots, a_n$. If A is a randomized algorithm, then we think of A as making coin tosses in order to choose each a_i. But Q must make each request σ_i with *no* knowledge of the previous answers $a_1, \ldots, a_{i-1}$. Thus, the "game" is not really very interesting — Q may as well choose all the σ_i right at the beginning. In order to make the game genuinely interactive, we consider stronger types of adversaries, which have access to the coin tosses of A.

Definition 5 *An* **adaptive adversary**, Q, *against a randomized algorithm,* A, *is an adversary which has access to* A*'s previous coin tosses. That is, when* Q *selects request* σ_i *it has knowledge of* A*'s responses* $a_1, a_2, \ldots, a_{i-1}$.

For an *adaptive* adversary, Q, however, we give Q the task of processing the *same* requests $\sigma_1, \sigma_2, \ldots, \sigma_n$ given the same resources and initial conditions as A. Thus Q incurs a cost, $C_{\mathrm{Q}}(\mathrm{A})$, which is the sum of the costs of Q's responses $q_1, q_2, \ldots, q_n$, even as A's cost, which we now call $C_{\mathrm{A}}(\mathrm{Q})$, is the sum of the cost of A's responses $a_1, a_2, \ldots, a_n$. We wish to define two distinct types of adaptive adversaries based on the way in which they process these requests and the resulting difference in *their* incurred cost, $C_{\mathrm{Q}}(\mathrm{A})$.

Definition 6 *An* **adaptive offline adversary**, Q, *against an algorithm,* A, *is an adversary which makes a request and then gets back* A*'s response before making its next request.* **After** A *has responded to all the requests* $\sigma_1, \sigma_2, \ldots, \sigma_n$, *then* Q *will process these same requests using the optimal* **offline** *strategy,* MIN. *Thus* $C_{\mathrm{Q}}(\mathrm{A}) = C_{\mathrm{MIN}}(\sigma)$, *where* σ *is the sequence of requests which* Q *makes.*

Definition 7 *An* **adaptive on-line adversary**, Q, *against an algorithm,* A, *is an adversary for which* both A and Q *must respond to each request before* Q *decides what the next request will be. The adversary,* Q, *learns of* A*'s response,* a_i, *to each request,* σ_i, *only* after *it has made its own response,* q_i, *but before it has to decide which request* σ_{i+1} *to make next.*

Remark 1: MIN is at least as good as any *on-line* strategy, so for any adaptive *on-line* adversary, there is an adaptive *offline* adversary which does *at least* as well. Thus the concept of an adaptive offline adversary is stronger than that of an adaptive on-line adversary.

Remark 2: The concept of an adaptive on-line adversary is stronger than that of an oblivious adversary.

Consider the case in which A is a randomized algorithm. Then $a_1, a_2, \ldots, a_n$ are random variables to be determined by A's coin tosses. Since $q_2, q_3, \ldots, q_n$ depend on the a_i's which precede them, they too are random variables. Similarly, the costs associated with each of these responses $a_1, a_2, \ldots, a_n, q_2, q_3, \ldots, q_n$ are random variables. Finally, the costs $C_{\mathrm{A}}(\mathrm{Q})$ and $C_{\mathrm{Q}}(\mathrm{A})$ are random variables, since they are formed by summing the costs of the a_i's and q_i's (including q_1), respectively. Thus in considering how effective A is we want to consider the expectations of these quantities, $E[C_{\mathrm{A}}(\mathrm{Q})]$ and $E[C_{\mathrm{Q}}(\mathrm{A})]$.

Definition 8 *A randomized algorithm,* A*, is α-competitive against an adaptive adversary,* Q*, if*

$$E[C_{\mathrm{A}}(\mathrm{Q})] \leq \alpha E[C_{\mathrm{Q}}(\mathrm{A})].$$

Remark 3: If A is deterministic, there is no difference between oblivious, adaptive offline, and adaptive on-line adversaries.

9 Some Results about Competitiveness against Various Types of Adversaries

An adaptive offline adversary is so strong that an algorithm which must confront an adaptive offline adversary is *not helped* by making use of randomization.

Theorem 12 *If there is a randomized on-line algorithm* A *which is α-competitive against any adaptive offline adversary, then there is an α-competitive deterministic on-line algorithm* G*.*

Proof:

Let A be distributed over deterministic on-line algorithms A_x. So for any adaptive off-line adversary Q,

$$E_x[C_{\mathrm{A}_x}(\mathrm{Q})] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x)] \leq 0.$$

Again, note the use of expectation in the second term. Q's cost is a random variable as well, based on A's coin tosses.

We now show how to construct a deterministic on-line algorithm G so that $C_{\mathrm{G}}(\sigma) \leq \alpha C_{\mathrm{MIN}}(\sigma)$, for any σ. Assume that the first request to A is σ_1; we can restrict our attention to adversaries that start with σ_1. Maximizing over such adversaries, we obtain:

$$Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q})] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x)]\} \leq 0.$$

We can now condition values on the answer a_1:

$$E_{a_1}[Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1]\}] \leq 0.$$

Since this expectation is non-positive, there must exist some specific a_1^* for which the value of the expression inside the brackets is non-positive. That is,

$$Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1^*] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1^*]\} \leq 0.$$

So our deterministic algorithm G will be one that plays a_1^* on request σ_1. Now Q plays σ_2; again, we restrict our attention to adversaries that play this way. Conditioning now on the choice of a_2,

$$E_{a_2}[Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1^*, a_2] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1^*, a_2]\}] \leq 0.$$

So there must be some answer a_2^* for which

$$Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1^* a_2^*] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1^* a_2^*]\} \leq 0,$$

the maximum being taken over the adversaries playing σ_1 first and playing σ_2 when the answer to σ_1 is a_1^*. The algorithm G will play a_2^*.

Proceeding in this way, we obtain a sequence of answers $a_1^*, \ldots, a_i^*$ for which

$$Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1^* \cdots a_i^*] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1^* \cdots a_i^*]\} \leq 0$$

until request σ_{i+1} is the special "STOP" request. At this point, the first term is the cost of our deterministic algorithm G on $\sigma = \sigma_1 \cdots \sigma_{i+1}$, and the second term is $C_{\mathrm{MIN}}(\sigma)$, since Q can answer the request sequence off-line.

Thus, by always playing an answer which makes the expectation non-positive, G will be α-competitive. □

Remark 4: This is merely a proof that G exists. It does not tell us how we might construct G, but the proof can be made somewhat constructive. We won't go into the details here.

Theorem 13 *If* G *is an α-competitive randomized on-line algorithm against any adaptive* on-line *adversary and there is a β-competitive randomized on-line algorithm, H, against any oblivious adversary, then* G *is an $\alpha\beta$-competitive randomized on-line algorithm against any adaptive* offline *adversary.*

Proof:

Let Q be any adaptive off-line adversary, and G_x, H_y denote the two randomized on-line algorithms with their associated probability distributions. We will also use the notation Q_x to emphasize that the behavior of Q depends on the coin tosses of G. We show a pair of inequalities which together will prove the theorem.

Consider first that Q is playing G_x. The algorithm H_y watches from a distance and plays along. Then Q is simply an oblivious adversary with respect to H_y, so for all x,

$$E_y[C_{\mathrm{H}_y}(\mathrm{Q}_x)] \le \beta C_{Q_x}(\mathrm{G}_x).$$

Thus we have the first of our inequalities:

$$E_x E_y[C_{\mathrm{H}_y}(\mathrm{Q}_x)] \le \beta E_x[C_{Q_x}(\mathrm{G}_x)]. \tag{2}$$

Now consider the case in which Q and H join forces to play against G. That is, we form an adaptive *on-line* adversary Q' which generates requests according to the strategy of Q and uses the answers $h_1, h_2, \ldots$ of H. The cost of Q' against G_x is $C_{\mathrm{H}_y}(\mathrm{Q}_x)$, since H_y is being used to provide the answers; meanwhile, the cost of G_x against Q' is simply $C_{\mathrm{G}_x}(\mathrm{Q}_x)$, since Q' uses Q to generate the requests. G is α-competitive against any adaptive on-line adversary, so for all y,

$$E_x[C_{\mathrm{G}_x}(\mathrm{Q}_x)] \le \alpha E_x[C_{\mathrm{H}_y}(\mathrm{Q}_x)]$$

$$E_y E_x[C_{\mathrm{G}_x}(\mathrm{Q}_x)] \le \alpha E_y E_x[C_{\mathrm{H}_y}(\mathrm{Q}_x)].$$

The left-hand side is independent of y, and we can swap the order of the expectations on this right-hand side, obtaining the second inequality:

$$E_x[C_{\mathrm{G}_x}(\mathrm{Q}_x)] \le \alpha E_x E_y[C_{\mathrm{H}_y}(\mathrm{Q}_x)]. \tag{3}$$

Combining these two inequalities,

$$E_x[C_{\mathrm{G}_x}(\mathrm{Q}_x)] \le \alpha\beta E_x[C_{Q_x}(\mathrm{G}_x)],$$

completing the proof that G is $\alpha\beta$-competitive against Q. □

Corollary 14 *Under the same assumptions, there is a* deterministic *$\alpha\beta$-competitive on-line algorithm.*

Example:

> For paging, there is a lower bound of k on the competitive ratio of a *deterministic* on-line algorithm, but there is also an H_k-competitive randomized on-line algorithm against any oblivious adversary. So by the contrapositive of the corollary, there is a lower bound of $\frac{k}{H_k}$ on the competitive ratio of a randomized on-line algorithm against any adaptive on-line adversary.

Corollary 15 *If* G *is an α-competitive randomized on-line algorithm against any adaptive* on-line *adversary, then there is an α^2-competitive deterministic on-line algorithm.*

The corollary is true because if G is α-competitive against any adaptive on-line adversary, then G is α-competitive against any oblivious adversary, and the corollary then follows from the theorem.

This last corollary is quite striking; it says that to show the existence of a competitive deterministic algorithm, we need only construct a randomized algorithm which is competitive against an adaptive on-line adversary.

10 Analysis of Random

In previous sections, we've seen Marking used as a competitive algorithm against an oblivious adversary. An even simpler randomized algorithm is Random, which, when needed, removes a uniformly selected page from the main algorithm's fast memory. We'll show

Theorem 16 (Raghavan and Snir [18]) Random *(*R*) is k-competitive against any adaptive on-line adversary* Q*.*

The proof of the theorem uses a *potential function* argument, which is a fundamental technique in establishing competitive ratios.

Proof:

The proof is based on the idea of a potential function Φ which measures the similarity between the k pages in Q's fast memory and the k pages in R's fast memory.

More precisely, let Q_i be the set of pages in Q's memory just after servicing request σ_i, and similarly for R_i. Then let $\Phi_i = |R_i \cap Q_i|$, the size of the intersection between Q's memory and R's.

Now, let $X_i = C_{\mathrm{R}}(\sigma_i) - kC_{\mathrm{Q}}(\sigma_i) - k(\Phi_i - \Phi_{i-1})$. The first thing to notice is that, for a sequence of requests $\sigma_1, \ldots, \sigma_j$,

$$\sum_{i=1}^{j} X_i = C_{\mathrm{R}}(\mathrm{Q}) - kC_{\mathrm{Q}}(\mathrm{R}) - k(\Phi_j - \Phi_0).$$

If we assume that all algorithms start with the same set of pages in fast memory, then $\Phi_0 = k$. Since no Φ_j can be greater than k, we have

$$\sum_{i=1}^{j} X_i \geq C_{\mathrm{R}}(\mathrm{Q}) - kC_{\mathrm{Q}}(\mathrm{R}),$$

from which, by the linearity of expectations,

$$\sum_{i=1}^{j} E[X_i] \geq E[C_{\mathrm{R}}(\mathrm{Q})] - kE[C_{\mathrm{Q}}(\mathrm{R})].$$

$$\cdots\ \underbrace{\sigma_b\ \cdots\ \rlap{$\overbrace{}^{R_{i-1}}$}\sigma_a\ \cdots}_{Q_{i-1}}\ \sigma_c\ \cdots\ \sigma_d\ \cdots$$

Figure 1: Pages of memory.

Thus, if we can establish that $\sum_{i=1}^{j} E[X_i] \leq 0$ then we will have shown that R is k-competitive.

Claim 17 $E[X_i] \leq 0$ *for all* X_i.

It is sufficient to show that the claim holds for any given pair of states Q_{i-1} and R_{i-1}:

$$E[X_i \mid Q_{i-1}, R_{i-1}] \leq 0 \qquad \text{for all states } Q_{i-1} \text{ and } R_{i-1}.$$

We will establish this by cases, considering where σ_i lies relative to Q_{i-1}, R_{i-1} and their intersection. For illustration, see Figure 1.

1. $\sigma_i \in (Q_{i-1} \cap R_{i-1})$
   The case is that of σ_a in Figure 1. Here there is no cost for either Q or R, since the page is already in both of their memories, so $C_{\mathrm{R}}(\sigma_i) = C_{\mathrm{Q}}(\sigma_i) = 0$. Since neither algorithm alters its memory, $\Phi_i = \Phi_{i-1}$, so $X_i = 0$, and thus
   $$E[X_i \mid Q_{i-1}, R_{i-1}] = 0.$$

2. $\sigma_i \in (Q_{i-1} \setminus R_{i-1})$
   This is like σ_b in Figure 1. In this case only R needs to do any work, so $C_{\mathrm{Q}}(\sigma_i) = 0$ and $C_{\mathrm{R}}(\sigma_i) = 1$. When R chooses the page to eliminate from fast memory it chooses it uniformly, choosing a page in Q_{i-1} with probability Φ_{i-1}/k. If the chosen page is in Q_{i-1} then one page will be eliminated from $Q_{i-1} \cap R_{i-1}$ and one added, so $\Phi_i = \Phi_{i-1}$. If not, then $\Phi_i = \Phi_{i-1} + 1$, since no page has been eliminated from the intersection. Thus
   $$E[\Phi_i - \Phi_{i-1} \mid Q_{i-1}, R_{i-1}] = 1 - \frac{\Phi_{i-1}}{k},$$
   so
   $$E[X_i \mid Q_{i-1}, R_{i-1}] = 1 - 0 - k\left(1 - \frac{\Phi_{i-1}}{k}\right) = 1 - k - \Phi_{i-1}.$$
   Since there was at least one page (σ_i) not in $Q_{i-1} \cap R_{i-1}$ we know that $\Phi_{i-1} < k$, so
   $$E[X_i \mid Q_{i-1}, R_{i-1}] \leq 0.$$

3. $\sigma_i \in (R_{i-1} \setminus Q_{i-1})$
   This is like σ_c in Figure 1. Since only Q needs to do any work, $C_{\mathrm{Q}}(\sigma_i) = 1$ and $C_{\mathrm{R}}(\sigma_i) = 0$. Now, whatever Q does in this case, the potential Φ_i will not decrease; as in the previous case, it will either stay the same or increase by one, so $\Phi_i - \Phi_{i-1} \geq 0$. Thus $X_i \leq 0 - k - k(0) < 0$, so
   $$E[X_i \mid Q_{i-1}, R_{i-1}] < 0.$$

4. $\sigma_i \notin (R_{i-1} \cup Q_{i-1})$
   This is the case of σ_d in Figure 1. Since both Q and R must pull σ_i into fast memory, $C_{\mathrm{Q}}(\sigma_i) = C_{\mathrm{R}}(\sigma_i) = 1$. Unlike the previous cases, Φ_i may here either stay the same, go up by one, or go down by one. Now, let's calculate the probability that Φ_i decreases by one. We have
   $$\Pr[\Phi_i - \Phi_{i-1} \geq 0 \mid Q_{i-1}, R_{i-1}] \geq \frac{1}{k}.$$
   To see this, fix the page that Q chooses to discard. If the page is not in $(Q_{i-1} \cap R_{i-1})$ then, with probability 1, Φ_i does not decrease. If the page is in $(Q_{i-1} \cap R_{i-1})$ then R will choose to discard the same page with probability $1/k$. Thus the probability in both cases is at least $1/k$. This gives us
   $$\Pr[\Phi_i - \Phi_{i-1} < 0 \mid Q_{i-1}, R_{i-1}] \leq 1 - \frac{1}{k},$$
   so
   $$E[\Phi_i - \Phi_{i-1} \mid Q_{i-1}, R_{i-1}] \geq (-1)(1 - \frac{1}{k}).$$
   This is enough to give us
   $$E[X_i \mid Q_{i-1}, R_{i-1}] \leq 1 - k(1) - k(-1)\left(1 - \frac{1}{k}\right) = 0.$$

$\diamond$

Since $E[X_i \mid Q_{i-1}, R_{i-1}] \leq 0$ for all Q_{i-1} and R_{i-1}, $E[X_i] \leq 0$, so
$$0 \geq \sum_{i=1}^{j} E[X_i] \geq C_{\mathrm{R}}(\mathrm{Q}) - kC_{\mathrm{Q}}(\mathrm{R})$$
which proves the theorem. $\square$

Note that we have assumed for this analysis that the adversary doesn't change its memory when it doesn't have to. Though we don't prove it here, it can be shown that this type of *lazy* adversary is at least as strong as any other, so the assumption is harmless.

11 The k–Server Problem

So far, we have been dealing with various concepts related to on-line algorithms, as well as the paging problem. In this and subsequent sections, we consider another famous problem for its apparent simplicity, the k-server problem. However, as it will appear, much less is known for the k-server problem than for the paging problem, although it has been extensively studied in the last few years.

The following is a practical analogy for the k–server problem. Suppose we have a city with a set of k police cars. When an emergency occurs somewhere in the city, one police car is chosen and dispatched to the site. We ignore the length of time for the car to travel to the site, ignore the length of time for the emergency to be handled, and count only the distance the police car must travel. Another emergency occurs and some car, maybe the same one, is chosen to travel to the next site. The emergencies must be attended to in order, and the dispatcher does not know in advance how many emergencies will occur where. Again, the cost function (to be minimized) is simply the sum of distances that the cars must travel, over all the emergencies. To formally analyze the k–server problem, we need to define a *metric space.*

Definition 9 *A* metric space *is a set of points V along with a distance function $d : (V \times V) \rightarrow \mathbb{R}$ such that*

1. $d(u, v) \geq 0, \ \ \forall u, v \in V.$
2. $d(u, v) = 0$ *iff* $u = v$
3. $d(u, v) = d(v, u), \ \ \forall u, v \in V$
4. $d(u, v) + d(v, w) \geq d(u, w), \ \ \forall u, v, w \in V.$

In other words, d is nonnegative, strictly positive between different points, satisfies the triangle inequality, and is symmetric. Often for purposes of this problem V is finite, but this is not obligatory.

Definition 10 (The k-server problem) *The input is a metric space V, a set of k "servers" located at points in V, and a stream of requests $\sigma_1, \sigma_2, \ldots$, each of which is a point in V. For each request, one at a time, you must move some server from its present location to the requested point. The goal is to minimize the total distance travelled by all servers over the course of the stream of requests.*

In general, the optimal solution to a k–server problem can be computed off-line by dynamic programming. (Dynamic programming can be done in polynomial time.) Another possible approach is to construct a graph such that the n requests and the k servers correspond to $n + k$ distinct vertices. For those familiar with the theory of network flows, the off-line problem can be solved as a minimum-cost flow problem on n copies of the graph. One must construct appropriate weighted edges. The class is invited to think about this as an exercise.

Claim 18 *For any stream of requests, on-line or off-line, only one server needs to be moved at each request.*

To show a contradiction, assume otherwise. In response to some request, σ_i, in your stream, you move server j to point σ_i and, in order to minimize the overall cost, you also move server k to some other location, perhaps to "cover ground" because of j's move. If server k is never again used, then the extra move is a waste, so assume server k is used for some subsequent request σ_m. However, by the triangle inequality (see the definition of a metric space), server k could have gone directly from its original location to the point σ_m at no more cost than stopping at the intermediate position after request σ_i.

We might, in general, consider algorithms which move more than one server at a time. While we will sometimes do this for purposes of analysis, we can in general ignore such algorithms thanks to the triangle inequality.

11.1 Special Cases of the k–Server Problem

1. Paging.
   The paging problem is a special case of the k-server problem, in which the k servers are the k slots of fast memory, V is the set of pages and $d(u, v) = 1$ for $u \neq v$. In other words, paging is just the k–server problem but with a uniform distance metric.

2. Two-headed Disk.
   You have a disk with concentric tracks. Two disk-heads can be moved linearly from track to track. The two heads are never moved to the same location and

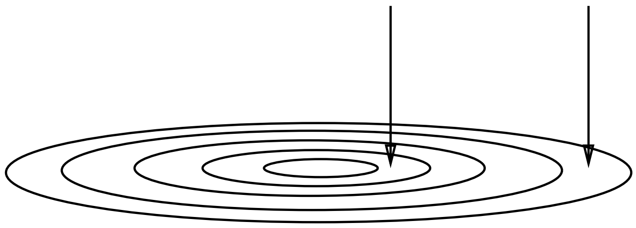


Figure 2: Two-headed disk.

   need never cross. The metric is the sum of the linear distances the two heads have to move to service all the disk's I/O requests. Note that the two heads move exclusively on the line that is half the circumference and the disk spins to give access to the full area.

11.2 Summary of known results

We already have results about paging that can be used for the k–server problem. Positive results about paging can't be generalized, but since paging is a case of the k–server problem, the lower bound for paging also applies in general.

Theorem 19 (Manasse-McGeoch-Sleator [16]) *For no metric spaces is there a deterministic α-competitive algorithm for the k-server problem such that $\alpha < k$.*

The general proof is similar to the previous proof with a uniform distance metric. However, in the case of paging we actually have a k-competitive deterministic on-line algorithm.

Open Problem: Whether or not there exists a k–competitive deterministic algorithm for the general k–server problem. In fact, for some time it was unknown whether there exists a $f(k)$-competitive algorithm, where $f(k)$ is a function of k.

Theorem 20 *Fiat, Rabani and Ravid [10] proved by induction on k that there is an $f(k)$-competitive algorithm for the general k–server problem, where $f(k) = e^{O(k \log k)}$.*

Very recently, it was shown by Koutsoupias and Papadimitriou [15] that the so-called *work-function algorithm* of Chrobak and Larmore [7] is $2k-1$-competitive. It is an open problem whether the same algorithm is in fact k-competitive.

11.2.1 Results for specific k

$k = 1$ This case is trivial, since there is never any choice about which server to send. Thus any reasonable algorithm is 1-competitive.

$k = 2$ There is a 2-competitive deterministic algorithm for this case, due to Manasse, McGeoch and Sleator [16]. Even with this small k, the algorithm is nontrivial and would make a challenging exercise. The work function algorithm is also known to be 2-competitive [7]. There has also been work on developing competitive algorithms for 2 servers which are "fast" in the sense that they perform only a constant amount of work per request [12, 4, 14].

$k = |V| - 1$. Then the algorithm Balance, described later, is k-competitive [16]. A variation of Balance is also known to be 10-competitive for the case $k = 2$ (Irani and Rubinfeld [12]).

11.2.2 Results for specific metric spaces

paging For paging, we've already seen a k-competitive algorithm.

line, tree For these configurations, there is a k-competitive deterministic algorithm [5].

circle For points on the circle, there is a deterministic k^3-competitive algorithm.

11.2.3 Greedy

The most obvious on-line algorithm for the k-server problem is GREEDY, in which a given request is serviced by whichever server is closest at the time. The following example, however, shows the major flaw in this algorithm:

Consider two servers 1 and 2 and two additional points A and b, positioned as follows (assume something like a Euclidean metric):

$$\bullet^1 \qquad\qquad\qquad\qquad \bullet^2 \quad \circ^a \quad \circ^b$$

Now take a sequence of requests $ababab\ldots$. GREEDY will attempt to service all requests with server 2, since 2 will always be closest to both A and b, whereas an algorithm which moves 1 to A and 2 to b, or vice versa, will suffer no cost beyond that initial movement. Thus GREEDY can't be α-competitive for any α.

11.2.4 The BALANCE Algorithm

We now describe the BALANCE algorithm mentioned above. At all times, we keep track of the total distance travelled so far by each server, D_{server}, and try to "even out" the workload among the servers. When request i arrives, it is serviced by whichever server, x, minimizes the quantity $D_x + d(x,i)$, where D_x is the distance travelled so far by server x, and $d(x,i)$ is the distance x would have to travel to service request i. BALANCE is k-competitive when $|V| = k+1$. However, it is not even competitive for $k = 2$. Consider indeed the following instance suggested by Jon Kleinberg. The metric space corresponds to a rectangle $abcd$ where $d(a,b) = d(c,d) = \alpha$ is much smaller than $d(b,c) = d(a,d) = \beta$. If the sequence of requests is $abcdabcd\ldots$, the cost of BALANCE is β per request, while the cost of MIN is α per request.

A slight variation of BALANCE in which one minimizes $D_x + 2d(x,i)$ can be shown to be 10-competitive for $k = 2$ [12].

12 The Randomized Algorithm, HARMONIC

While GREEDY doesn't work very well on its own, the intuition of sending the closest server can be useful if we randomize it slightly. Instead of sending the closest server every time, we can send a given server with probability inversely proportional to its distance from the request.

Thus for a request A we can try sending a server at x with probability $1/(Nd(x,a))$ for some N. Since, if On is the set of on-line servers we want

$$1 = \sum_{x \in \mathrm{On}} \frac{1}{Nd(x,a)}$$

we set

$$N = \sum_{x \in \mathrm{On}} \frac{1}{d(x,a)}$$

This algorithm is known as the HARMONIC algorithm (H). Note, once again, that in the special case of paging this is identical to RANDOM.

HARMONIC is competitive as stated below.

Theorem 21 (Grove [11]) HARMONIC *is* $(\frac{5}{4}k2^k - 2k)$*-competitive against an adaptive on-line adversary.*

Before this fairly recent result, HARMONIC was only known to be 3^{17000}-competitive for $k = 3$ [3] and 3-competitive for 2 servers [6].

There is a known lower bound for this algorithm. Specifically, for some adaptive on-line adversary Q, we know that $C_H(\mathrm{Q}) = \binom{k+1}{2} C_{\mathrm{Q}}(\mathrm{H})$, (where H denotes HARMONIC) so we can't hope to do better than $\binom{k+1}{2}$-competitiveness. It is open whether the competitive factor for HARMONIC is indeed equal to this lower bound.

Finally, recall that we can get an α^2-competitive deterministic on-line algorithm from an α-competitive randomized algorithm. So this theorem will also give us a $(\frac{5}{4}k2^k - 2k)^2$-competitive deterministic algorithm, although the algorithm that we get this way is far from efficient.

12.1 Analysis of HARMONIC

We prove here a slightly weaker result than Grove's result.

Theorem 22 HARMONIC *is* $k^2 2^{k-1}$*-competitive against an adaptive online adversary.*

Proof:

In the following, let OFF denote both the servers of the online adaptive adversary which HARMONIC plays against, as well as their locations in the metric space.

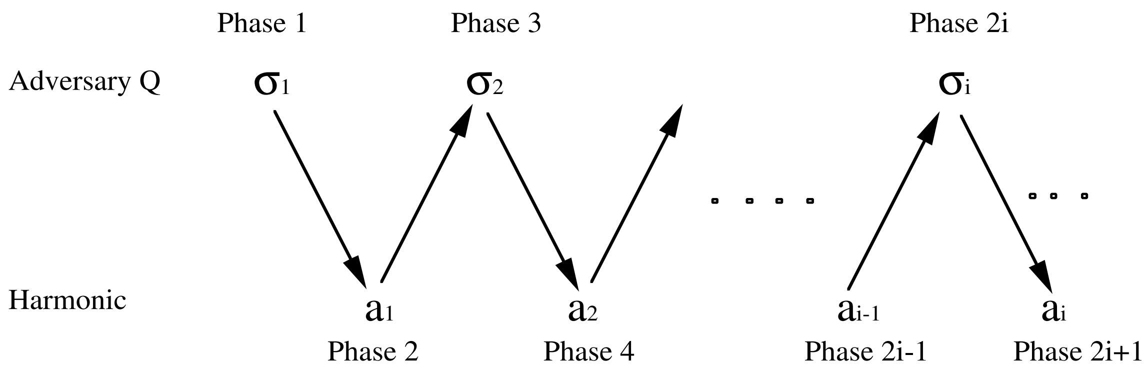


Figure 3: Odd and even phases of OFF vs. ON.

We define phases (Figure 3) by:

- In the odd phase $2i-1$, the online adversary moves a server to vertex A at cost d_i. The adversary then makes a request σ_i at vertex A. (Note that OFF can move its server before making the request because it is an online adversary, so the order makes no difference).

- In the even phase $2i$, the player moves an online server $a_i \in$ ON to A and incurs a cost of l_i. l_i is in fact a random variable since HARMONIC chooses the server to move probabilistically.

The actions of ON and OFF taken during one pair of odd/even phases are illustrated in Figure 4.

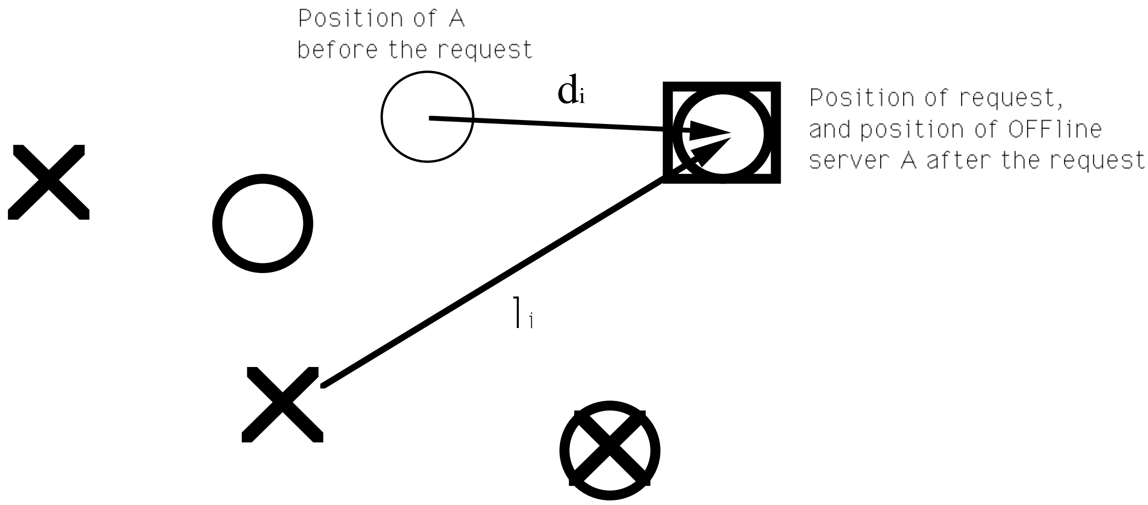


Figure 4: The movement of servers. X's denote the servers of ON while O's denote the servers of OFF.

12.1.1 Analysis by Means of a Potential Function

To aid in the analysis of HARMONIC, we introduce a potential function Φ. Let $\Phi(j)$ denote the value of Φ at the end of phase j. The potential function will be required to satisfy the following properties.

1. $\Phi(0) = 0$, $\Phi(j) \geq 0$.

2. $\Phi(2i-1) - \Phi(2i-2) \leq k^2 2^{k-1} d_i$.

3. $E[l_i + \Phi(2i) - \Phi(2i-1)] \leq 0$.

These properties allow us to prove the competitive ratio of $k^2 2^{k-1}$ since, by summing 2. and 3., we obtain:

$$
\begin{aligned}
E[\textstyle\sum_{i=1}^{j} l_i] &= \sum_{i=1}^{j} E[l_i] \\
&= \sum_{i=1}^{j}(E[l_i + E[\Phi(2i) - \Phi(2i-1)]])g + \sum_{i=1}^{j} E[\Phi(2i-1) - \Phi(2i-2)] \\
&\quad -E[\Phi(2j)] + E[\Phi(0)] \\
&\leq k^2 2^{k-1} \sum_{i=1}^{j} E[d_i] \\
&= k^2 2^{k-1} E[\sum_{i=1}^{j} d_i].
\end{aligned}
$$

12.1.2 Derivation of the Potential Function Φ

We now obtain a potential function satisfying the above constraints. Φ will depend on three factors:

1. The vertices in ON.
2. The vertices in OFF.
3. The past history of moves by ON and OFF.

Φ is obtained by means of a matching M mapping the vertices in ON bijectively to the vertices in OFF. If $x \in \mathrm{ON}$ then we denote by $M(x) \in \mathrm{OFF}$ the server in OFF matched to x.

Initially we impose that $x = M(x)$ for all servers x in ON (since ON and OFF start with the same initial configuration of servers). This implies that $\Phi(0) = 0$.

In an odd phase (i.e. when a server from OFF is moved, the matching is kept unchanged.

When (during an even phase) a server in ON is moved from vertex b to the position of the OFFline server A (where a request occurs and with $M(a) = A$), the matching is updated by the following rule: If ON moves the server b which was originally matched to server B from OFF, then M is updated according to $M(a) = B$ and $M(b) = A$.

We can now define Φ by:

$$\Phi = \sum_{x \in \mathrm{ON}} \lambda(x)$$

where for $\Gamma \subseteq \mathrm{OFF}$ we define the radius by:

$$R(x, \Gamma) = \max_{Y \in \Gamma} d(x, Y)$$

and

$$\lambda(x) = k2^k \min_{\Gamma \subseteq \mathrm{OFF}} \left\{ \frac{R(x,\Gamma)}{2^{|\Gamma|}} : M(x) \in \Gamma \right\}.$$

Also define Γ_x^* to be the argmin of the term in the definition of $\lambda(x)$.

Note that $\lambda(x) \geq kd(x, M(x))$ for all x since $2^{|\Gamma|} \leq 2^k$ and $R(x,\Gamma) \geq d(x, M(x))$ for any $\Gamma \ni M(x)$.

Two final definitions are:

$$\rho(x) = R(x, \Gamma_x^*) \text{ and } n(x) = |\Gamma^*|.$$

So $\lambda(x) = k2^k \frac{\rho(x)}{2^{n(x)}}$.

12.1.3 Why we Argue on Matchings

A small example now suffices to provide an intuitive feeling for why matchings provide a tool for measuring the potential difference between ON and OFF. Suppose that there are seven servers positioned as shown in Figure 5. If the adversary consistently makes requests to the poorly matched server on the right, then the adversary will pay 0 cost, while Harmonic will keep paying until it moves a server from the left side of the network. In the matching, one edge will be large, and so both $R(x, \Gamma_x^*)$ and $\lambda(x)$ will be large until such a move is made.

Figure 5: An example of a configuration of OFF's servers (O's) and ON's servers (X's). If the adversary requests at the unmatched OFFline server on the right, then Harmonic eventually pays a high cost while the adversary pays 0.

12.1.4 Proof of Properties about Φ

Property 1

Initially $\Gamma_x^* = \{x\}$ so $\rho(x) = 0$ which means $\lambda(x) = 0$ and $\Phi(0) = 0$. It is clear that $\Phi(j) \geq 0$.

Property 2

In an odd phase, the adversary moves a server in OFF by a distance d_i. Let $\lambda'(x)$ be the value of λ after the move. Then

$$\lambda'(x) \leq k2^k \frac{R'(x, \Gamma_x^*)}{2^{\Gamma_x^*}}$$

where Γ_x^* is the value from before the move and:

$$\begin{array}{rcl} R'(x,\Gamma_x^*) & = & \max_{Y\in\Gamma_x^*} d'(x,Y) \\ & \leq & \max_{Y\in\Gamma_x^*}\{d(x,Y)\} + d_i \\ & = & R(x,\Gamma_x^*) + d_i. \end{array}$$

Note that the distance function is primed to indicate that Y may have moved during the phase. However, the matching is not altered during an odd phase, so we were able to use the same Γ_x^* (since $M(x)$ still belongs to Γ_x^*).

Since $|\Gamma_x^*|$ is at least one, we now can calculate that:

$$\begin{array}{rcl} \lambda'(x) & \leq & k2^k \dfrac{R'(x,\Gamma_x^*)}{2^{|\Gamma_x^*|}} \\ & \leq & \lambda(x) + k2^k \dfrac{d_i}{2^{|\Gamma_x^*|}} \\ & \leq & \lambda(x) + k2^{k-1} d_i. \end{array}$$

Therefore,

$$\begin{array}{rcl} \Phi(2i-1) - \Phi(2i-2) & = & \sum_{x\in \mathrm{ON}}(\lambda'(x) - \lambda(x)) \\ & \leq & k(k2^{k-1}d_i) \\ & \leq & k^2 2^{k-1} d_i. \end{array}$$

Property 3

We now need to show that $E[l_i + \Phi(2i) - \Phi(2i-1)] \leq 0$. In the rest of this proof, we shall be conditioning on P, the current position of the servers in ON and OFF. Clearly, if the inequality is true while conditioning on P, then it will remain true when we take the expectation over P. First consider what $E[l_i|P]$ represents. It is the expected distance that the server moved by Harmonic will have to travel. By the definition of Harmonic, we know that, given P, $x \in$ ON is moved with probability $\frac{1}{Nd(x,A)}$.

Therefore,

$$\begin{array}{rcl} E[l_i|P] & = & \sum_{x\in \mathrm{ON}} \frac{1}{Nd(x,A)} d(x,A) \\ & = & \frac{k}{N} \end{array}$$

since ON contains k servers.

So for Φ to satisfy property 3, we must show that

$$E[\Phi(2i) - \Phi(2i-1)|P] \leq -E[l_i|P] = -\frac{k}{N}.$$

Because we are in an even phase, OFF already has a server at A. We will examine the changes in the matching depending on which server ON moves in order to approach this problem. Each of the cases has a probability attached, and so the results of each case can be used to determine an expected value for $\Phi(2i) - \Phi(2i-1)$. Note that in the following analysis the servers of OFF are fixed, since the adversary has already moved its server during the odd phase.

Case 1 (occuring with probability $\frac{1}{Nd(a,A)}$):

ON moves server a to the request at A (where $M(a) = A$ before the phase).

Then after the move, $\lambda'(a) = 0$ and $\lambda'(c) = \lambda(c)$ for all $c \neq a$. Therefore,

$$\Phi(2i) - \Phi(2i-1) = -\lambda(a) = -k2^k \frac{\rho(a)}{2^{n(a)}}.$$

Case 2:

ON moves a server $b \neq a$ where $M(b) = B$ before the move.

Case i: $d(a, B) \leq \rho(a)$.

Then $\lambda'(c) = \lambda(c)$ for $c \neq a, c \neq b$. Moreover, $\lambda'(b) = 0$. And lastly, $\lambda'(a) \leq \lambda(a)$ since the fact that $d(a, B) \leq \rho(a)$ implies that B (the new OFFline server matched to a, belongs to Γ_a^*) (see Figure 6). Therefore, $\Phi(2i) - \Phi(2i-1) \leq 0$.

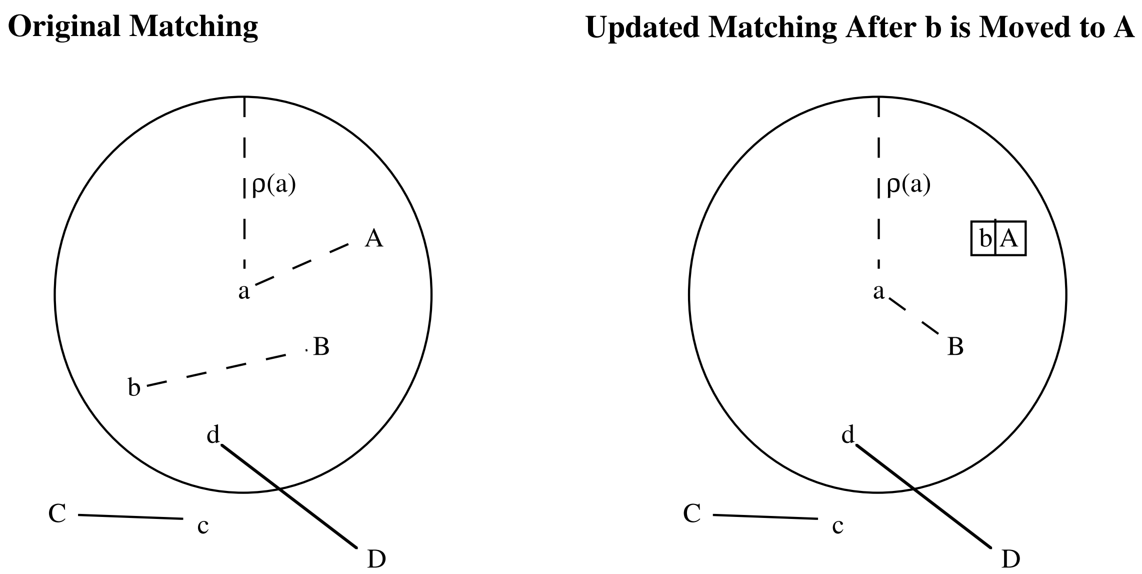


Figure 6: The configuration of the matching when $d(a, B) \leq \rho(a)$, and when the matching is updated.

Case ii: $d(a, B) > \rho(a)$.

Now define l so that B is the lth closest OFF server to a from outside of the disk of radius $\rho(a)$ centered at a. Consider a disk centered at a of radius $\rho(a) + d(A, b) + \rho(b)$. (See Figure 7). Clearly B is in that disk, since

$$d(a, B) \leq d(a, A) + d(A, b) + d(b, B) \leq \rho(a) + d(A, b) + \rho(b).$$

In fact, this big disk contains at least $\max(n(a)+l, n(b))$ servers from OFF. (i.e. $n(a)+l$ is all OFF servers in the disk centered at a with radius $d(a,B)$ and $n(b)$ is all servers in the disk of radius $\rho(b)$ centered at b).

Therefore,

$$\begin{aligned}
\lambda'(a) &\leq k2^k \frac{R(a,\Gamma)}{2^{|\Gamma|}} \\
&\leq k2^k \frac{\rho(a) + d(A,b) + \rho(b)}{2^{\max(n(a)+l, n(b))}} \\
&\leq k2^k \left[\frac{\rho(a)}{2^{n(a)}} + \frac{d(A,b)}{2^{n(a)+l}} + \frac{\rho(b)}{2^{n(b)}}\right] \\
&= \lambda(a) + k2^k \frac{d(A,b)}{2^{n(a)+l}} + \lambda(b).
\end{aligned}$$

Furthermore, $\lambda'(b) = 0$ since b is moved to the position A of the request and $M'(b) = A$. Lastly, $\lambda'(c) = \lambda(c)$ for $c \neq a, c \neq b$. This means that in this case when b is the server moved to A:

$$\Phi(2i) - \Phi(2i-1) \leq \frac{k2^k d(A,b)}{2^{n(a)+l}}.$$

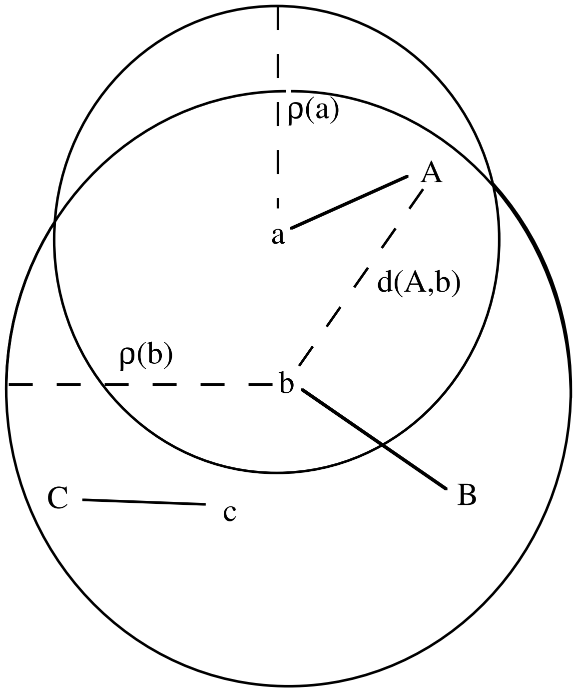


Figure 7: The configuration of the matching when $d(a,B) > \rho(a)$.

12.1.5 The Expected Value for $\Phi(2i) - \Phi(2i-1)$ Over All Cases

Weighting the three cases presented above by the probability of their occurences, we have that:

$$\begin{aligned}
E[\Phi(2i) - \Phi(2i-1)] &\leq \frac{1}{Nd(a,A)}\left(-\frac{k2^k\rho(a)}{2^{n(a)}}\right) + \sum_{l=1}^{k-n(a)} \frac{1}{Nd(A,b_l)}\left(\frac{k2^k d(A,b_l)}{2^{n(a)+l}}\right) \\
&\leq \frac{k2^k}{N2^{n(a)}}\left[-1 + \sum_{l=1}^{k-n(a)} \frac{1}{2^l}\right] \\
&= -\tfrac{k}{N} \text{ (geometric series)}
\end{aligned}$$

where b_l is the server which before the move has $M(b) = B$, where B is the lth closest OFF server to a from outside the disk of radius $\rho(a)$ centered at a. Note that the sum is to $k - n(a)$ which is the number of OFF servers outside of the disk of radius $\rho(a)$ centered at a.

This is the result we wanted.

□

12.1.6 The Intuition

Recall that $\lambda(x) = k2^k \min_{\Gamma} \dfrac{R(x,\Gamma)}{2^{|\Gamma|}} = k2^k \dfrac{\rho(A)}{2^{n(a)}}$.

Here is some intuition behind the use of the use of the exponential factor $2^{|\Gamma|}$. If $n(a)$ is large then there are not many OFF servers outside of $\rho(a)$ so the $2^{n(a)}$ term is large, making $\lambda(a)$ smaller. On the other hand, if $n(a)$ is small, then there are many OFF servers outside of the circle, and these contribute to a big potential function by making $\frac{2^k}{2^{n(a)}}$ big. These two cases correspond to the cost we could expect. If $n(a)$ is large, then HARMONIC will have a reasonably high probability of moving an ONline server whose matched OFFline server is in the disk centered at a of radius $\rho(a)$ to cover a request at A. This would result in a "small" expected cost. However, by a similar argument, we would expect a larger expected cost if $n(a)$ is small.

13 A k-competitive, deterministic algorithm for trees

The results of this section are due to Chrobak and Larmore [5]. Let (V, E) be a tree (an undirected graph without cycles) with a positive distance function on each edge. We view each edge, e, as an actual segment of length $d(e)$. Let W denote the infinite set of all points on all edges in E, and let requests be any points in W. (The cost of travelling from an endpoint of e to a non-vertex point in edge e will simply be the fraction of $d(e)$ proportional to the location of the point from the end.) The algorithm presented will of course also apply to the discrete case where all requests are vertices.

Definition 11 *At any time, there is some request σ which we are trying to service. We say that a server is* active *if there is no server located between it and σ. If there are several servers located at the same point, with no servers located between them and σ, we pick exactly one of them to be active. We can do this deterministically according to some ordering of the servers if we like, calling the highest-priority server the active one. Note that this definition makes sense because the (acyclic) tree structure means that there is exactly 1 path between any two points.*

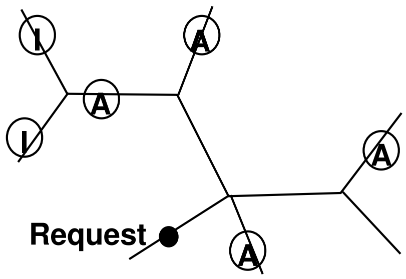


Figure 8: Active servers are marked with an A, inactive ones with an I. Note that if all servers move with a constant speed, the closest one (at the bottom) will reach the request and the others will become inactive somewhere on the way to the request.

Our algorithm A is simple to describe: to service a request i, all active servers move towards i at constant speed, each one stopping when one of the following is true

1. the server reaches the destination, i.
2. the server is eclipsed by another server and becomes inactive;

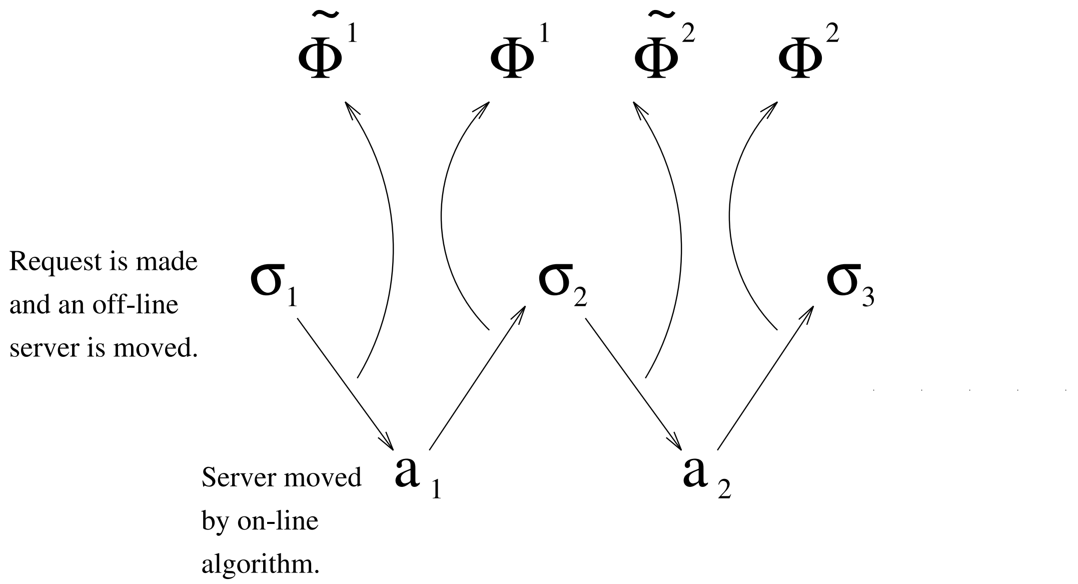
According to the second condition, a server that is active at the beginning of a request might not be active later on in the request. As soon as a server sees another server between it and i, it becomes inactive and stops moving.

13.1 Proof of k-competitiveness

We will use a potential function to show that, for any sequence $\sigma_1, \sigma_2, \ldots$ of requests, $C_A(\sigma) \leq kC_{\text{MIN}}(\sigma)$. Let Φ^t be the value of the potential function after the on-line algorithm has processed σ_t. Let $\tilde{\Phi}^t$ be the value of the potential function before the on-line algorithm has processed σ_t but after the off-line algorithm has processed it. Suppose that the on-line servers are at postions $s_1, s_2, \ldots, s_k$ and the off-line servers are at positions $a_1, a_2, \ldots, a_k$. We define the potential funtion by

$$\Phi = \sum_{i<j} d(s_i, s_j) + kM(s, a).$$

where $M(s, a)$ is the minimum cost matching between the on-line servers and the off-line servers. i.e. $M(s, a) = \min_\tau \sum_{i=1}^{k} d(a_i, s_{\tau(i)})$ where the minimum is taken over all permutations τ. Note that this is a potential function which often arises in the analysis of algorithms for the k-server problem.



Claim 23

$$\begin{aligned}\tilde{\Phi}^t - \Phi^{t-1} &\leq kC_{\text{MIN}}(\sigma_t),\\ \Phi^t - \tilde{\Phi}^t &\leq -C_A(\sigma_t).\end{aligned}$$

where A is the on-line algorithm. From this claim we can derive that $\Phi^t - \Phi^{t-1} \leq kC_{\text{MIN}}(\sigma_t) - C_A(\sigma_t)$. If we take the sum over t then the left hand side telescopes to give us $\Phi^{\text{final}} - \Phi^0 \leq kC_{\text{MIN}}(\sigma) - C_A(\sigma)$. We know that $0 \leq \Phi^{\text{final}}$ and so $C_A(\sigma) \leq kC_{\text{MIN}}(\sigma) + \Phi^0$. We assume that all of the on-line servers and all of the off-line servers are initially at the same single point. This implies that $\Phi^0 = 0$ and so $C_A(\sigma) \leq kC_{\text{MIN}}(\sigma)$. From this inequality we can see that the algorithm is k-competitive. It only remains for us to prove the claim.

Proof of claim We first consider what happens when the off-line algorithm moves a server to the request. Choose τ so that $M(s,a) = \sum_{i=1}^{k} d(a_i, s_{\tau(i)})$. Suppose that the off-line algorithm chooses to move a_l to the request. Since none of the on-line servers are moving $\sum_{i<j} d(s_i, s_j)$ does not change. Hence

$$\begin{aligned}\tilde{\Phi}^t - \Phi^{t-1} = k\Delta M(s,a) &\leq k(\text{distance traveled by } a_l \text{ to the request})\\ &= kC_{\text{MIN}}(\sigma_t).\end{aligned}$$

We now consider how Φ changes when the on-line algorithm moves its servers. Number the on-line servers so that $s_1, s_2, \ldots, s_q$ are active and $s_{q+1}, \ldots, s_k$ are inactive.

The active servers are all moving at the same speed towards the request. Suppose that they all move a small distance ϵ. It is easy to see that without loss of generality a_l is matched to an active server in the minimum cost matching. (Recall that a_l is the off-line server which is already at the request.) Hence the distance between these two servers decreases by ϵ. Also, the distance between other pairs of matching servers increases by at most ϵ. Therefore,

$$\Delta M \leq (q-1)\epsilon - \epsilon = (q-2)\epsilon.$$

We also have to consider the change in $\sum_{i<j} d(s_i, s_j)$. Let

$$\begin{aligned} S_I &= \sum_{q<i<j} d(s_i, s_j), \\ S_{IA} &= \sum_{i\leq q<j} d(s_i, s_j), \\ S_A &= \sum_{i<j\leq q} d(s_i, s_j). \end{aligned}$$

Then,

$$\begin{aligned} \Delta S_A &= -2\epsilon \begin{pmatrix} q \\ 2 \end{pmatrix} = 2\epsilon \frac{q(q-1)}{2} = -\epsilon q(q-1), \\ \Delta S_{IA} &= (k-q)(\epsilon - (q-1)\epsilon) = (k-q)\epsilon(2-q), \\ \Delta S_I &= 0. \end{aligned}$$

The first equation comes from the fact that each pair of active servers move towards each other by a distance -2ϵ. The second is true because there are $k-q$ inactive servers, each of which has one active server moving away from it and $q-1$ moving towards it. From the above equations we derive that $\sum_{i<j} d(s_i, s_j)$ increases by $\epsilon\,[(k-q)(2-q) - q(q-1)]$. Hence

$$\begin{aligned} \Delta\Phi &\leq \epsilon\,[(k-q)(2-q) - q(q-1) + k(q-2)] \\ &= -q\epsilon. \end{aligned}$$

Now $q\epsilon$ is the cost incurred by the on-line algorithm when it moves its q active servers a distance ϵ. Hence $\Phi^t - \tilde{\Phi}^t \leq -C_A(\sigma_t)$ which is the inequality that we were trying to prove.

13.2 Paging as a case of k-server on a tree

Note that a special case of the k-server problem on a tree is paging. Create M tree-vertices corresponding to the pages of main (slow) memory, create one dummy tree-vertex, v, and connect v to all the other M vertices using edges of length $\frac{1}{2}$. Note that the cost of moving a server from one page to another is $\frac{1}{2} + \frac{1}{2}$, the cost of

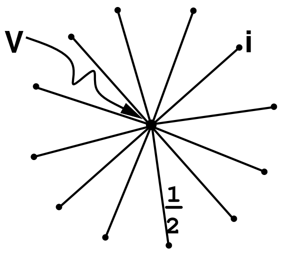


Figure 9: Paging as a special case of the k-server problem on a tree.

swapping. (More generally, we could let the length of the edge from v to page i be any positive function $f(i)$, obtaining the *Generalized Paging* problem, where the cost of swapping pages i and j is $f(i) + f(j)$.)

Let us consider the behavior of the above algorithm A on this special case. The resulting algorithm for paging is known as FLUSH-WHEN-FULL. The interpretation is simple if one keeps track of "marked" pages. When a server is at a vertex corresponding to a page p, this page is considered marked. As soon as the server leaves that vertex to go towards v, the page will be unmarked.

Initially there are k servers on k pages. These pages are thus marked. Suppose request σ_i causes the first page fault. Algorithm A will then move all servers towards v, resulting in the unmarking of all pages in fast memory. All these servers move at constant speed towards v and thus will reach the middle vertex v at the same time. One (arbitrarily selected) server at v will continue to the requested page σ_i and the other $k-1$ will become inactive. The page σ_i will then be marked. Later, if there is a page fault on say σ_j and there is at least one server at vertex v, one of these active servers will be moved to σ_j. In terms of paging, this is interpreted as swappping σ_j with an arbitrarily selected *unmarked* page of fast memory and then marking σ_j. The claim of the previous section implies that FLUSH-WHEN-FULL is k-competitive.

Of course we don't have to move more than one server per step. We could "pretend" to moves servers simultaneously but actually just keep track of where the servers should be (by keeping track of which pages of fast memory are marked) and move one server to the request-destination per step. The cost per step would be the the total distance travelled by that server since the last time it reached a request-destination. This way the cost per page-fault is always exactly one.

FLUSH-WHEN-FULL is much like MARKING, except that MARKING uses randomization to select a server at a tie. FLUSH-WHEN-FULL is k-competitive while MARKING is H_k-competitive against an oblivious adversary. This shows how useful a simple randomization step can be. FLUSH-WHEN-FULL applies to *Generalized Paging* and is the only known k-competitive algorithm for that problem.

Question: Could LRU be an example of FLUSH-WHEN-FULL? Yes, it is. In other words, LRU would never get rid of a marked page and thus, by carefully selecting which unmarked page to remove from fast memory, FLUSH-WHEN-FULL reduces to

LRU. To see that a marked page is never removed from fast memory by LRU, notice that each marked page has been requested after the last request to any unmarked page.

14 Electric Network Theory

We will use electric network theory for a randomized k-server algorithm due to Coppersmith, Doyle, Raghavan, and Snir [8]. Their algorithm will be k-competitive against an adaptive on-line adversary for a subclass of metrics.

An *electric network* is a graph $G = (V, E)$ such that each edge has weight $\sigma(e) = \frac{1}{R(e)}$, where $R(e) \in \mathbb{R}^+$ is called the *resistance* and $\sigma(e) \in \mathbb{R}^+$ is called the *conductance* of edge e. We can then ask what the *effective* resistance (also called the *equivalent* resistance) between any two vertices is, i.e. the resistance which is felt when applying a difference of voltage between any two vertices. The effective conductance is the inverse of the effective resistance.

For resistances in series, the effective resistance between the endpoints is equal to the sum of the resistances. For resistances in parallel, the effective conductance is equal to the sum of the conductances. See Figures 10 and 11. In general, though, these rules are not enough to determine the effective resistance between any two vertices in an electric network (consider the case when the underlying graph is the complete graph K_4 on 4 vertices). In full generality, one has to use Kirchoff's first law and the relation $V = RI$. Simply stated, the first law says that the sum of the currents entering a junction is equal to the sum of the currents leaving that junction.

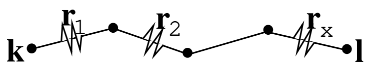


Figure 10: Resistance in series. The effective resistance between k and l is $r_1 + r_2 + \ldots + r_x$.

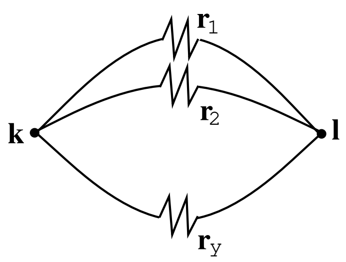


Figure 11: Resistances in parallel. The effective conductance between k and l is $\frac{1}{r_1} + \frac{1}{r_2} + \ldots + \frac{1}{r_y}$.

Definition 12 *A distance matrix $D = [d_{ij}]$ is* resistive *if it is the effective resistance matrix of some electric network G, $G = (V, E)$.*

Now let us use the effective resistance as a metric. This is justified by the following proposition.

Proposition 24 *If D is resistive then D is symmetric and D satisfies the triangle inequality $d_{ij} + d_{jk} \geq d_{ik}$ $\forall i, j, k$.*

The converse of this proposition is not necessarily true. A metric does not necessarily induce a resistive distance matrix. In fact, there are metric spaces on four points that aren't resistive. Satisfying the triangle inequality isn't enough.

Metrics which correspond to resistive distance matrices will be referred to as *resistive.*

What are resistive matrices or resistive metrics? Here are two simple examples.

1. if D is 3×3, symmetric, and satisfies the triangle inequality, then it is always resistive. Hence, the above proposition is true in the converse for 3×3 matrices. As an example, consider a triangular network with effective resistances 3, 4, and 5. See figure 12. We claim that the edge-resistances are $\frac{3}{11}$, $\frac{2}{11}$ and $\frac{1}{11}$ (see figure). We verify it for the vertices 1 and 3. Consider the two paths between these vertices. We need the effective resistance to be 4, and thus the effective conductance to be $\frac{1}{4}$. Verify that

$$\frac{2}{11} + \left(\frac{1}{\frac{11}{3} + \frac{11}{1}} \right) = \frac{11}{44} = \frac{1}{4}$$

2. a tree metric is resistive. To see this, make a tree of resistances with $R_{ij} = d(i, j)$ along the edges of the tree. Because there are no cycles, every pair of points is connected by a series of resistances. The effective resistance is the sum of the edge-resistances.

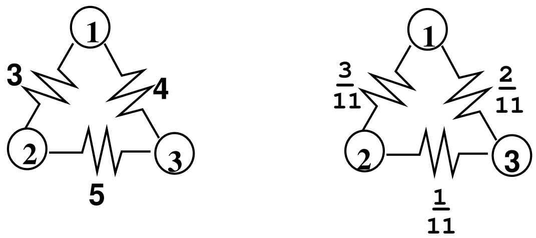


Figure 12: The given matrix of effective resistances (*left*) is relabelled (*right*) as the corresponding electrical network with resistances.

Another property of resistive metrics is given by the following lemma.

Lemma 25 *If D is resistive then any induced submatrix D' is resistive.*

How can we compute an array of conductances σ from D? We will not go through the proof here, but the outline of the algorithm is as follows:

- Assume D is $n \times n$. Construct an $(n-1) \times (n-1)$ matrix $\bar{D}$ such that $\bar{d}_{ij} = (d_{1i} + d_{1j} - d_{ij})/2 \geq 0$ for $2 \leq i, j \leq n$.
- Let $\bar{\sigma} = \bar{D}^{-1}$.
- Let $\sigma_{ij} = -\bar{\sigma}_{ij}$ for $i \neq j$, $2 \leq i, j \leq n$.
- Let $\sum_{j \neq i} \sigma_{ij} = \bar{\sigma}_{ii}$ for $2 \leq i \leq n$. This allows us to determine σ_{i1} and σ_{1i}.
- Let $\sigma_{ii} = 0$ for $1 \leq i \leq n$.

If D is resistive, the above procedure yield a matrix σ of non-negative conductances.

14.1 The Algorithm: RWALK

Consider the following algorithm for the k-server problem. We have servers at $a_1 \ldots a_k$, and we get a request at a_{k+1}. Consider the distance matrix D' among these $k+1$ elements. Assuming D' is resistive, calculate σ' and then move the server at a_i to the request with probability

$$Pr(a_i \rightarrow a_{k+1}) = \frac{\sigma'_{i,k+1}}{\sum_{1 \leq j \leq k} \sigma'_{j,k+1}}$$

Notice that a shorter distance corresponds to a higher conductance which in turn corresponds to a higher probability. Thus, this algorithm is intuitively correct because we are more likely to move a server "close" to the request.

Theorem 26 *If every induced subgraph on $k+1$ points is resistive, then* RWALK *is k-competitive against an adaptive on-line adversary.*

Two important cases covered by this theorem are:

- k-server with $k = 2$. RWALK can be used and is thus 2-competitive.
- k-server on a tree. We know that D' will always be resistive (by Lemma 25) and hence RWALK can be used. The theorem shows that RWALK is k-competitive in this case.

Proof:

We need to show

$$E[C_{\text{RWALK}}(\text{Q})] \le kE[C_{\text{Q}}(\text{RWALK})]$$

where Q is an adaptive on-line adversary. We can rewrite this as

$$E[C_{\text{RWALK}} - kC_{\text{Q}}] \le 0.$$

We will in fact show that

$$E[\Phi + C_{\text{RWALK}} - kC_{\text{Q}}] \le 0$$

where Φ is a potential function that we will define.

We will show that at every step,

$$E[\Delta\Phi + C_{\text{RWALK}}(step) - kC_{\text{Q}}(step)] \le 0.$$

In words, this means that, in any single step, the cost of RWALK is at most k times the cost of the adversary, once the costs are amortized according to the potential function Φ. Summing over all steps, we obtain that

$$E[\Phi_n - \Phi_0 + C_{\text{RWALK}}(\text{Q}) - kC_{\text{Q}}(\text{RWALK})] \le 0,$$

for a sequence of n requests. Since $\Phi_n \ge 0$ and we shall assume that $\Phi_0 = 0$, we derive the competitive factor of k. ($\Phi_0 = 0$ corresponds to the case in which all servers start initially from the same location (see below), and $\Phi_0 \neq 0$ would just result in a weak competitive factor of k.)

We want Φ to measure the difference between the locations of RWALK's servers and Q's servers. Let $a = a_1, \ldots, a_k$ be the set of locations of RWALK's servers, and let $b = b_1, \ldots, b_k$ be the set of locations of Q's servers. Then define a potential function

$$\Phi(a, b) = \sum_{1 \le i < j \le k} d(a_i, a_j) + kM(a, b)$$

where $M(a, b)$ is the cost of the minimum cost matching between A and b (in other words, $M(a, b) = \min_\alpha \sum_{i=1}^{k} d(a_i, b_{\alpha(i)})$ where the minimum is taken over all permutations α). Intuitively, the sum is the amount of "separation" of the elements of A, and the matching term is the "difference" between the algorithm and the adversary. Some intuitive argument justifying the $M(a, b)$ term goes as follows. If the algorithm has to pay much more than the adversary, it means that (some of) the algorithm's servers are far away from the adversary's servers, implying that $M(a, b)$ is large. The reduction in $M(a, b)$ can therefore be used to pay for the move.

Let us consider the request σ_i. We decompose the processing of σ_i into two steps.

- step 1: adversary moves a server to σ_i.

- step 2: on-line algorithm RWALK moves a server to σ_i.

Step 1. The adversary moves from b_j to b'_j. Then $\Delta\Phi \leq kd(b_j, b'_j)$ because given the minimum matching for the old A and b, we can get a matching for the new A and b by using the same matching. So if b_j was matched to a_i, then

$$\Delta\Phi \leq k(d(a_i, b'_j) - d(a_i, b_j)) \leq kd(b_j, b'_j).$$

We also have

$$C_{\text{RWALK}}(step) = 0,$$

and

$$C_{\text{Q}}(step) = d(b_j, b'_j).$$

Therefore,

$$E[\Delta\Phi - kC_{\text{Q}}(step)] \leq 0.$$

Case 2. The on-line algorithm moves. Since the adversary has already moved a server to the requested node, assume WLOG that the request is at b_1 (we can always renumber). Let the minimum matching M before the move be $(a_1, b_1)\ldots(a_k, b_k)$ (again we can renumber to make this the case).

Assume that RWALK moves a_j to the request b_1. We claim that

$$\Delta M(a, b) \leq d(a_1, b_j) - d(a_1, b_1) - d(a_j, b_j)$$

since a possible matching can be defined from the minimum cost matching between the old A's and b's by simply assigning a_1 to b_j and a_j to b_1:
$a_1 - b_j$
$a_2 - b_2$
$\vdots$
$a_j - b_1$
$\vdots$
$a_k - b_k$
From the triangle inequality, we have $d(a_1, b_j) - d(a_j, b_j) \leq d(a_1, a_j)$ and thus $\Delta M(a, b) \leq d(a_1, a_j) - d(a_1, b_1)$. Therefore,

$$\Delta\Phi \leq \sum_{i \neq j}[d(a_i, b_1) - d(a_i, a_j)] + k[d(a_1, a_j) - d(a_1, b_1)].$$

We also have $C_{\text{RWALK}}(step) = d(a_j, b_1)$ and hence

$$\begin{aligned}
\Delta\Phi + C_{\text{RWALK}}(step) &\leq \sum_{i \neq j}[d(a_i, b_1) - d(a_i, a_j)] + k[d(a_1, a_j) - d(a_1, b_1)] + d(a_j, b_1) \\
&\leq \sum_{i}[d(a_i, b_1) - d(a_i, a_j) + d(a_1, a_j) - d(a_1, b_1)].
\end{aligned}$$

We will show that $E[\Delta\Phi + C_{\mathrm{RWALK}}(step)] \le 0$ by showing that

$$\sum_{j=1}^{k} \sigma'_{j,k+1} \left(\sum_{i=1}^{k} e(a_1, a_i, a_j, b_1) \right) = 0$$

where

$$e(a_1, a_i, a_j, b_1) = d(a_i, b_1) - d(a_i, a_j) + d(a_1, a_j) - d(a_1, b_1).$$

For convenience, define $a_{k+1} = b_1$. Now we have to recall a few facts about electric networks to get a handle on the $\sigma'_{j,k+1}$. Recall that

$$\sigma'_{j,k+1} = -\bar{\sigma}_{j,k+1} \;\; \text{for } j = 2, ..., k$$

and

$$\sum_{j=1}^{k} \sigma'_{j,k+1} = \bar{\sigma}_{k+1,k+1}.$$

We know that $\bar{D} \cdot \bar{\sigma} = I$, so

$$\sum_{j=2}^{k+1} \bar{d}(a_i, a_j)\ \bar{\sigma}_{j,k+1} = 0 \;\; \text{if } i = 2, ..., k.$$

Summing over i, we get

$$\sum_{i=2}^{k} \sum_{j=2}^{k+1} \bar{d}(a_i, a_j)\ \bar{\sigma}_{j,k+1} = 0$$

or, using the definition of $\bar{\sigma}$,

$$\sum_{i=2}^{k} \left(-\sum_{j=2}^{k} \bar{d}(a_i, a_j)\sigma'_{j,k+1} + \sum_{j=1}^{k} \sigma'_{j,k+1}\bar{d}(a_i, a_{k+1}) \right) = 0.$$

Since $\bar{d}(a_i, a_1) = (d(a_i, a_1) - d(a_1, a_1) - d(a_i, a_1))/2 = 0$, we can extend the first summation over j from 1 to k:

$$\sum_{i=2}^{k} \sum_{j=1}^{k} \left(\bar{d}(a_i, b_1) - \bar{d}(a_i, a_j) \right) \sigma'_{j,k+1} = 0.$$

The term in parenthesis is

$$\begin{aligned} \bar{d}(a_i, b_1) - \bar{d}(a_i, a_j) &= \frac{1}{2}[d(a_1, a_i) + d(a_1, b_1) - d(a_i, b_1) - d(a_1, a_i) - d(a_1, a_j) + d(a_i, a_j)] \\ &= -\frac{1}{2} e(a_1, a_i, a_j, b_1) \end{aligned}$$

so we finally get

$$\sum_{j=1}^{k} \sigma'_{j,k+1} \left(\sum_{i=1}^{k} e(a_1, a_i, a_j, b_1) \right) = 0$$

since we can do the sum of i from 1 because $e(a_1, a_1, a_j, b_1) = 0$.

Therefore, we have that the expected amortized cost is negative and thus we have a k-competitive algorithm for any resistive metric.

□

15 The work function algorithm

In this section, we show that the work function algorithm for the k-server problem is $2k-1$-competitive. The algorithm was proposed by Chrobak and Larmore [7] but they were only able to show that it is 2-competitive when $k = 2$. The following proof is due to Koutsoupias and Papadimitriou [15]. Before this result was obtained, no algorithm was known to have a competitive ratio which was polynomial in k for an arbitrary metric space. It is believed that the work function algorithm is in fact k-competitive.

15.1 Definition of the work function

Let $A_0 = \{a_{0,1}, a_{0,2}, \ldots, a_{0,k}\}$ be the initial configuration of the servers and let $r_1, r_2, \ldots, r_t, \ldots$ be the sequence of requests. If X is a multiset of k points in the metric space we define the *work function* at time t by,

$$\begin{aligned} w_t(X) = &\text{the optimal cost of servicing the first } t \text{ requests} \\ &\text{and ending up with the servers in configuration } X. \end{aligned}$$

From this definition we can see that the initial work function w_0 can be expressed as,

$$\begin{aligned} w_0(X) &= \text{minimum cost matching between } A_0 \text{ and } X \\ &= \min_{\text{perm } \sigma} \sum_{i=1}^{k} d(a_{0,i}, x_{\sigma(i)}) \\ &= d(A_0, X). \end{aligned}$$

Before servicing r_t the servers will be in some configuration Y. Hence,

$$w_t(X) = \min_{Y : r_t \in Y} [w_{t-1}(Y) + d(Y, X)].$$

We shall need a number of properties of the work function.

Lemma 27

$$\forall t, X, Z : w_t(Z) \leq w_t(X) + d(X, Z).$$

Proof:

$$\begin{aligned} w_t(X) + d(X, Z) &= \min_{Y : r_t \in Y} \{w_{t-1}(Y) + d(Y, X) + d(X, Z)\} \\ &\geq \min_{Y : r_t \in Y} \{w_{t-1}(Y) + d(Y, Z)\} \\ &= w_t(Z). \end{aligned}$$

□

Lemma 28

$$\forall t, X : w_t(X) = \min_{x \in X} \{w_{t-1}(X - x + r_t) + d(r_t, x)\}.$$

Proof:

- ($\leq$) This is obvious since if we take $Y = X - x + r_t$ then $d(Y, X) = d(r_t, x)$.
- ($\geq$) Take the optimum Y in the definition of $w_t(X)$. Let x be the element of X which is matched to r_t in the minimum cost matching between X and Y. Let $Y' = X - x + r_t$. Then,

$$\begin{array}{rcll} w_{t-1}(Y) + d(Y, X) & = & w_{t-1}(Y) + d(Y, Y') + d(x, r_t) & \\ & \geq & w_{t-1}(Y') + d(x, r_t) & \text{by lemma 27} \\ & \geq & \min_{x \in X} \left(w_{t-1}(X - x + r_t) + d(x, r_t)\right). & \end{array}$$

□

Lemma 29

$$\forall t, X : w_t(X) \geq w_{t-1}(X).$$

Proof:

$$\begin{array}{rcll} w_t(X) & = & w_{t-1}(Y) + d(Y, X) & \text{for some } Y \\ & \geq & w_{t-1}(X) & \text{by lemma 27.} \end{array}$$

□

Lemma 30

$$\forall X : r_t \in X \Rightarrow w_t(X) = w_{t-1}(X).$$

Proof:

From lemma 28 with $x = r_t$ we get $w_t(X) \leq w_{t-1}(X)$. Together with lemma 29 this implies the result. □

Lemma 31

$$\forall t, X : w_t(X) = \min_{x \in X} \{w_t(X - x + r_t) + d(r_t, x)\}.$$

Proof:

This is immediate from lemmas 28 and 30. □

15.2 Definition of the work function algorithm

Let A_{t-1} be the configuration of the on-line servers before r_t is requested. When the request comes in the work function algorithm moves $a \in A_{t-1}$ which minimizes

$$w_{t-1}(A_{t-1} - a + r_t) + d(a, r_t).$$

i.e. $A_t \leftarrow A_{t-1} - a + r_t$ where

$$a := \operatorname{argmin}_{a \in A_{t-1}} \left\{ w_{t-1}(A_{t-1} - a + r_t) + d(a, r_t) \right\}.$$

Note that $w_{t-1}(A_{t-1} - a + r_t) + d(a, r_t) = w_t(A_{t-1})$. Hence if we were to move the servers to A_{t-1} after serving request r_t then we would move the server at r_t to a. The work function algorithm does this move backwards.

For analysis purposes we assume that the on-line servers and the off-line servers finish at the same position. (If this is not the case we can repeatedly request points where there is an off-line server but no on-line one. This does not increase the off-line cost.) It is clear from the above definition that the on-line cost of serving r_t is $d(a, r_t)$. If r_f is the final request then the total off-line cost of serving all the requests is,

$$w_f(A_f) = \sum_{t=1}^{f} \left[w_t(A_t) - w_{t-1}(A_{t-1}) \right].$$

We define the off-line pseudocost of serving request r_t to be $w_t(A_t) - w_{t-1}(A_{t-1})$. We then define the extended cost of serving request r_t to be the sum of the off-line pseudocost and the on-line cost. i.e.

$$\text{extended cost } = w_t(A_t) - w_{t-1}(A_{t-1}) + d(a, r_t).$$

But, by definition of the work function algorithm, $w_t(A_{t-1}) = w_{t-1}(A_t) + d(a, r_t)$. Hence,

$$\begin{aligned} \text{extended cost} &= w_t(A_{t-1}) - w_{t-1}(A_{t-1}) \\ &\leq \max_X \left\{ w_t(X) - w_{t-1}(X) \right\}. \end{aligned}$$

Note that the above equation gives an upper bound for the extended cost which is *independent of the algorithm* or the location of the servers. The definition of extended cost means that if

$$\text{extended cost } \leq (c+1) \text{ off-line cost}$$

then the work-function algorithm is c-competitive. We shall show that

$$\text{extended cost } \leq 2k \,(\text{ off-line cost}),$$

proving the desired competitive ratio.

Our next goal is to define a potential function $\Phi(w)$. We shall then show that, at every request, the increase of the potential function is lower bounded by the extended cost and that the total increase of the potential function is upper bounded by $2k$ times the off-line cost. More formally, we will show that

$$\begin{array}{lcl} \Phi(w_t) - \Phi(w_{t-1}) & \geq & \max_X \{w_t(X) - w_{t-1}(X)\}, \\ \Phi(w_f) & \leq & 2kw_f(A_f) + c. \end{array}$$

For this purpose, we shall need a better understanding of $\max_X \{w_t(X) - w_{t-1}(X)\}$. The following lemma is useful. Koutsoupias and Papadimitriou show a generalization of this lemma which they call "quasiconvexity".

Lemma 32

$$\forall t, \forall A, B, \forall a \in A, \exists b \in B : w_t(A) + w_t(B) \geq w_t(A - a + b) + w_t(B - b + a).$$

Proof:

By induction on t. When $t = 0$, $w_t(X) = w_0(X) = d(A_0, X)$ for all configurations X. Let a_0 be the element of A_0 which is matched with a in the minimum cost matching between A and A_0. Let b be the element of B which is matched with a_0 in the minimum cost matching between A_0 and B. Then,

$$\begin{array}{rcl} w_0(A) + w_0(B) & = & d(A_0, A) + d(A_0, B) \\ & = & d(A_0 - a_0, A - a) + d(A_0 - a_0, B - b) + d(a_0, a) + d(a_0, b) \\ & \geq & d(A_0, A - a + b) + d(A_0, B - b + a) \\ & = & w_0(A - a + b) + w_0(B - b + a). \end{array}$$

Now assume that the lemma is true for $t - 1$. Then by lemma 28,

$$(4) \qquad \left\{ \begin{array}{lcl} \exists x \in A : w_t(A) & = & w_{t-1}(A - x + r_t) + d(x, r_t), \\ \exists y \in B : w_t(B) & = & w_{t-1}(B - y + r_t) + d(y, r_t). \end{array} \right.$$

If $a = x$ then let $b = y$. By definition of w_t,

$$\begin{array}{lcl} w_t(A - x + y) & \leq & w_{t-1}(A - x + r_t) + d(y, r_t), \\ w_t(B - y + x) & \leq & w_{t-1}(B - y + r_t) + d(x, r_t). \end{array}$$

Hence,

$$w_t(A - x + y) + w_t(B - y + x) \leq w_t(A) + w_t(B).$$

If $a \neq x$ then $a \in A - x + r_t$. Using the inductive hypothesis on $A - x + r_t$ and $B - y + r_t$, $\exists b \in B - y + r_t$:

(5)

$$w_{t-1}(A - x + r_t) + w_{t-1}(B - y + r_t) \geq w_{t-1}(A - x + r_t - a + b) + w_{t-1}(B - y + r_t - b + a)$$

But,

$$(6) \qquad \left\{ \begin{array}{lcl} w_t(A - a + b) & \leq & w_{t-1}(A - a + b - x + r_t) + d(x, r_t), \\ w_t(B - b + a) & \leq & w_{t-1}(B - b + a - y + r_t) + d(y, r_t). \end{array} \right.$$

Finally, (4), (5) and (6) imply the lemma. □

Definition 13 *A is called a minimizer of the point a with respect to w_t if A minimizes $(w_t(A) - \sum_{x \in A} d(a,x))$.*

We can redefine $\max_X \{w_t(X) - w_{t-1}(X)\}$ as

$$\max_X \left\{ (w_t(X) - \sum_{x \in X} d(x, r_t)) - (w_{t-1}(X) - \sum_{x \in X} d(x, r_t)) \right\}.$$

A minimizer of r_t with respect to w_{t-1} would thus maximize the second term. The next lemma shows somewhat surprisingly that such a minimizer would in fact maximize the entire expression.

Lemma 33 *Assume that A is a minimizer of r_t with respect to w_{t-1}. Then $w_t(A) - w_{t-1}(A) = \max_X \{w_t(X) - w_{t-1}(X)\}$.*

Proof:

We need to show that,

$$\forall B : w_t(A) + w_{t-1}(B) \geq w_{t-1}(A) + w_t(B).$$

This is equivalent to,

$$\forall a \in A : w_{t-1}(A - a + r_t) + d(a, r_t) + w_{t-1}(B) \geq \min_{b \in B} \{w_{t-1}(B - b + r_t) + d(b, r_t) + w_{t-1}(A)\}.$$

By assumption,

$$w_{t-1}(A) - \sum_{x \in A} d(x, r_t) \leq w_{t-1}(A - a + b) - \sum_{y \in A - a + b} d(y, r_t),$$

or,

$$w_{t-1}(A) + d(b, r_t) \leq w_{t-1}(A - a + b) + d(a, r_t).$$

Hence we only need to show that,

$$w_{t-1}(A - a + r_t) + w_{t-1}(B) \geq \min_{b \in B} \{w_{t-1}(B - b + r_t) + w_{t-1}(A - a + b)\}.$$

But by lemma 32, $r_t \in A - a + r_t \Rightarrow \exists b \in B$:

$$w_{t-1}(A - a + r_t) + w_{t-1}(B) \geq w_{t-1}(A - a + b) + w_{t-1}(B - b + r_t).$$

$\square$

Lemma 34 *Assume that A is a minimizer with respect to w_{t-1}. Then A is also a minimizer with respect to w_t.*

Proof:

We need to show that,

$$\forall B : w_t(A) - \sum_{a\in A} d(a, r_t) \geq w_t(B) - \sum_{b\in B} d(b, r_t),$$

or, for all $y \in B$,

$$\min_{x\in A}\left\{w_{t-1}(A - x + r_t) + d(x, r_t) - \sum_{a\in A} d(a, r_t)\right\} \leq w_{t-1}(B-y+r_t)+d(y, r_t)-\sum_{b\in B} d(b, r_t).$$

Since A is a minimizer of r_t with respect to w_{t-1},

$$w_{t-1}(A) - \sum_{a\in A} d(a, r_t) \leq w_{t-1}(B - x + y) - \sum_{b\in B+x-y} d(b, r_t),$$

or,

$$w_{t-1}(A) + d(x, r_t) - \sum_{a\in A} d(a, r_t) \leq w_{t-1}(B + x - y) - \sum_{b\in B} d(b, r_t) + d(y, r_t).$$

Now $r_t \in B - y + r_t$, hence by lemma 32,

$$\exists x \in A : w_{t-1}(A - x + r_t) + w_{t-1}(B - y + x) \leq w_{t-1}(A) + w_{t-1}(B - y + r_t),$$

which implies that

$$w_{t-1}(A - x + r_t) + d(x, r_t) - \sum_{a\in A} d(a, r_t) \leq w_{t-1}(B - y + r_t) + d(y, r_t) - \sum_{b\in B} d(b, r_t).$$

□

15.3 Definition of the potential function

We define the potential function as follows.

$$\Phi(w_t) = \min_U \left\{ k w_t(U) + \sum_{i=1}^{k} \min_{B_i}\left(w_t(B_i) - \sum_{j=1}^{k} d(u_j, b_{ij})\right)\right\}.$$

Thus B_i is a minimizer of u_i with respect to w_t.

Lemma 35 *We can assume that $r_t \in U$ when trying to minimize the above expression.*

Proof:

By lemma 31,

$$w_t(U) = w_t(U - u_l + r_t) + d(r_t, u_l) \text{ for some } u_l.$$

By the triangle equality,

$$d(r_t, u_l) - d(u_l, b_{ij}) \geq -d(r_t, b_{ij}). \tag{7}$$

Let $U' = U - u_l + r_t$.

$$\Phi(w_t) = \min_{U'} \begin{cases} kw_t(U') & + \sum_{i=1, i\neq l}^{k} \min_{B_i} \left(w_t(B_i) - \sum_{j=1}^{k} d(u_i, b_{ij}) \right) \\ & + \min_{B_l} \left(w_t(B_l) + \sum_{j=1}^{k} \left(-d(u_l, b_{lj}) + d(r_t, u_l) \right) \right). \end{cases} \tag{8}$$

But by (7),

$$-\sum_{j=1}^{k} \left(d(u_l, b_{lj}) + d(r_t, u_l) \right) \geq -\sum_{j=1}^{k} d(r_t, b_{lj}).$$

Substituting this inequality into (8) gives us the required result. □

Let $U, B_1, B_2, \ldots, B_k$ attain the minimum in the definition of $\Phi(w_t)$. The above lemma means that we can assume $r_t = u_l$. Hence we can assume that B_l is a minimizer of r_t with respect to w_t. Therefore by lemma 34 we can take B_l to be a minimizer of r_t with respect to w_{t-1}. Let $A = B_l$. By defeinition of Φ, we derive that

$$\Phi(w_{t-1}) \leq kw_{t-1}(A) + \sum_{i=1}^{k} \left(w_{t-1}(B_i) - \sum_{j=1}^{k} d(u_i, b_{ij}) \right).$$

Therefore, by Lemmas 29 and 32, we obtain

$$\Phi(w_t) - \Phi(w_{t-1}) \geq w_t(A) - w_{t-1}(A) = \max_{X} \left\{ w_t(X) - w_{t-1}(X) \right\}.$$

Summing over all requests, we obtain that the total extended cost is at most $\Phi(w_f) - \Phi(w_0)$.

We only need to upper bound this quantity in terms of the off-line cost. Let $U = A_f, B_i = A_f$ (where A_f is the final location of the off-line or on-line servers). This gives us,

$$\begin{aligned} \Phi(w_f) & \leq kw_f(A_f) + kw_f(A_f) - \sum_{a \in A_f, b \in A_f} d(a, b) \\ & \leq 2kw_f(A_f). \end{aligned}$$

Hence if we let $c = \Phi(w_0)$ (constant dependent only on the initial configuration, not the request sequence) then,

$$\text{total extended cost} \leq 2kw_f(A_f) + c.$$

By the remarks following the definition of extended cost this means that the work function algorithm is $(2k-1)$-competitive.

References

[1] L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5:78–101, 1966.

[2] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:2–14, 1990.

[3] P. Berman, H. Karloff, and G. Tardos. A competitive three-server algorithm. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, 1990.

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

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

[6] M. Chrobak and L. Larmore. HARMONIC is 3-competitive for 2 servers. *Theoretical Computer Science*, 98:339–346, 1992.

[7] M. Chrobak and L. Larmore. The server problem and on-line games. In D. Sleator and L. McGeoch, editors, *On-line algorithms*, volume 7 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 11–64. AMS, 1992.

[8] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. Random walks on weighted graphs, and application to on-line algorithms. *Journal of the ACM*, 40:421–453, 1993. A preliminary version appeared in the Proceedings of the 22nd STOC, 369–378, 1990.

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

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

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

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

[13] D. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.

[14] J. M. Kleinberg. On-line algorithms for robot navigation and server problems. Master's thesis, MIT, Cambridge, MA, 1994.

[15] 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.

[16] M. Manasse, L. McGeoch, and D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.

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

[18] P. Raghavan and M. Snir. Memory vs. randomization in on-line algorithms. In *Proceedings of the 1989 ICALP Conference*, 1989.

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

[20] A. Yao. New algorithms for bin packing. *Journal of the ACM*, 27:207–227, 1980.