Alcom-FT Technical Report Series
ALCOMFT-TR-02-28

More on weighted servers
or
FIFO is better than LRU *

Leah Epstein† Csanád Imreh‡ Rob van Stee§

May 2, 2002

Abstract

We consider a generalized 2-server problem on the uniform space in which servers have different costs. Previous work focused on the case where the ratio between these costs was very large. We give results for varying ratios. For small ratios, we present an optimal algorithm which is trackless. We present a general lower bound for trackless algorithms depending on the cost ratio, proving that our algorithm is the optimal trackless algorithm up to a constant factor for any cost ratio. The results are extended for the case where we have two sets of servers with different costs.

1 Introduction

The weighted k-server problem was introduced by Fiat and Ricklin [7]. In this problem, we are given a metric space $M = (A, d)$ with k mobile servers, and each server s_i has a weight $x_i > 0$. At each step, a request $r \in M$ is issued that has to be served by one of the servers by moving to r. The cost for server s_i to serve request r is $d(r, s_i) \cdot x_i$. This problem is an on-line problem: each time that there is a request, it needs to be served before the next request becomes known. We denote the cost of an algorithm ALG on a request sequence σ by $\text{ALG}(\sigma)$. We denote an optimal off-line algorithm that knows all the input in advance by OPT. The goal of an on-line algorithm $\mathcal{A}$ is to minimize its competitive ratio $\mathcal{R}(\mathcal{A})$, which is defined as the smallest value $\mathcal{R}$ that satisfies

$$\mathcal{A}(\sigma) \leq \mathcal{R} \cdot \text{OPT}(\sigma) + c,$$

for any request sequence σ and some constant c (independent of σ).

*This research has been partially supported by the Israel Science Foundation GRANT 250/01, by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the Hungarian National Foundation for Scientific Research, Grant TO30074.

†School of Computer Science, The Interdisciplinary Center, P.O.B 167, 46150 Herzliya, Israel. lea@idc.ac.il.

‡Max-Planck Institut für Informatik, Stuhlsatzenhausweg 85, Saarbrücken, Germany, on leave from University of Szeged, Hungary. imreh@mpi-sb.mpg.de.

§Institut für Informatik, Albert-Ludwigs-Universität, Georges-Köhler-Allee, 79110 Freiburg, Germany. vanstee@informatik.uni-freiburg.de.

In [7] a doubly exponential upper bound in k is given for the uniform spaces, furthermore for the special case, where only two weights are allowed a $k^{O(k)}$ competitive algorithm is presented. They also show that the competitive ratio is at least $k^{\Omega(k)}$ in any space with at least $k+1$ points. In [10], a simple upper bound of kw_{arg}/w_{min} is proven for the general case, where w_{min} is the minimal and w_{arg} is the average weight of the servers.

The special case of two servers and uniform spaces was investigated in [6]. There a 5-competitive work function based algorithm and matching lower bound, and a 5-competitive memoryless randomized algorithm with matching lower bound are presented.

All previous work (expect the simple result of [10]) focus on the asymptotic cases where the ratio between the weights of the servers tends to ∞. We consider instead the case of smaller ratios and obtain the surprising result that for the weighted 2-server problem in a uniform space, an algorithm that uses both its servers equally is optimal as long as the ratio between the weights is at most 2.2.

We also consider the more general case, where we have k servers with speed 1 and k servers with speed x. Since we only investigate uniform spaces, the problem can also be seen as a caching problem where we have two caches of size k: the cheap cache C, and the expensive cache E. This type of cache is called *a two-level cache* [1, 5]. For this reason, we borrow some terminology from caching theory. We formulate our problem as follows. The algorithm has to serve a sequence of requests for pages. If the requested page is not in the caches, then we have to put it into one of the caches, evicting a page from the chosen cache if it is full. This event is called a fault. The set of possible pages is called the slow memory. Moving a page into C has cost 1, and moving a page into E has a cost of $x \geq 1$.

All of the previous algorithms for the weighted server problem store information about most of the appearing request points: the algorithm SAMPLE of [7] has a counter of the points, in the work function algorithm of [6] this information is coded in the work function. This yields that these algorithms might have an extremely large space requirement for large number of different requested points. For the original k-server problem a class of algorithms called trackless is introduced in [3] to avoid this problem. In the case of trackless algorithms, for each request point the algorithm is only given as input the distances of the current server positions to the request point. The algorithm may memorize such distance values, but it is restricted from storing explicitly any points of the metric space.

For the special case of uniform spaces the trackless property changes into the rule that the algorithm is not allowed to use bookmarks in the slow memory to distinguish between the pages. This means that in the case of a fault the decision that which cache is used for the requested page and which page is removed from this cache must be independent on the requested page itself. This restriction seems to be very strong at first look, but we must note that for paging the optimal deterministic marking algorithms (see [8]) are trackless, and even for the more general web caching problem the optimal algorithm (see [12]) is trackless. Randomized trackless algorithms for the paging problem are investigated in [2]. It is shown that it is necessary to use bookmarks to reach the optimal $\log k$ competitive ratio.

The algorithm which is optimal for small x in the case $k = 1$ is trackless and is based on the well-known paging algorithm FIFO. We also analyse the modified version of the other well-known paging algorithm LRU, and we obtain that in the weighted case FIFO is better, then LRU. We show that for trackless algorithms, the competitive ratio must grow linearly as a function of kx, proving that the presented algorithm is an optimal trackless algorithm up to a constant factor for arbitrary x. For the sake of completeness we also show a simple upper bound for general k, (this upper bound is the special case of the upper bound from [10]), pointing out that the marking algorithms are optimal trackless algorithms up to a constant factor 2.

2 Marking algorithms

To begin with, we give a simple proof for the competitiveness of marking algorithms, which is a special case of the bound from [10].

The marking algorithms, which are defined in [8], work in phases. At the beginning of each phase, all pages in the caches are unmarked. When a page from one of the caches is requested we mark it. If there is a fault, then we evict some unmarked page (determined by the algorithm), load the requested page and mark it. The phase is ended, when we have a fault and every page in the caches is marked. We unmark the pages and the new phase is started.

Theorem 2.1 *All marking algorithms have a competitive ratio of at most* $k(1+x)$.

Proof. To prove the upper bound note that every marking algorithm has a cost of exactly $k(1+x)$ per phase, whereas OPT has a cost of at least 1 per phase. □

3 Two servers

It is possible to adapt the well-known paging algorithms FIFO and LRU for the current problem, by considering the two caches as one big cache and ignoring the difference in costs. We denote these adaptations also by FIFO and LRU.

Theorem 3.1 $\mathcal{R}(\text{FIFO}) = \max(2, 3(1+x)/4)$.

Definition 3.1 *A* relevant *request sequence is a request sequence on which* FIFO *faults on every request.*

We can map any request sequence σ onto a relevant request sequence σ' by removing all requests on which FIFO does not fault. This does not affect the cost of FIFO or its decisions: a request in σ' is put into the cheap cache by FIFO if and only if FIFO puts the corresponding request in σ in the cheap cache. Hence $\text{FIFO}(\sigma') = \text{FIFO}(\sigma)$. Moreover, $\text{OPT}(\sigma') \leq \text{OPT}(\sigma)$. Therefore we only need to consider relevant request sequences to determine the competitive ratio of FIFO. Note that for a relevant request sequence, a marking phase always consists of two consecutive requests.

Lemma 3.1 *Suppose* $\{a_i\}_{i=1}^{\ell}$ *is a relevant sequence and* OPT *faults on some request* a_j. *Then* OPT *faults on at least one of the requests* a_{j+1} *and* a_{j+2}.

Proof. By definition of FIFO, any three consecutive requests (in a relevant sequence) must be to distinct pages. If OPT does not fault on a_{j+1}, then it has a_j and a_{j+1} in its cache after the request for a_j. Thus OPT must fault on a_{j+2}, since that is a different page. □

Definition 3.2 *A relevant interval is a subsequence $\{a_i\}_{i=j_1}^{j_2}$ of a request sequence with the following property:* OPT *faults on $a_j \iff (j - j_1) \bmod 2 = 0$.*

By Lemma 3.1 we can partition any relevant sequence into relevant intervals. Note that all relevant intervals end with a fault by OPT, and have odd length.

Lemma 3.2 *For $f \geq 1$,* OPT *has at least f expensive faults on a relevant interval of length at least $4f + 1$.*

Proof. Consider a relevant interval of length at least 5 and partition it into f subintervals of length 4, and one final subinterval of length at most 3. Consider any subinterval of length 4. It has the form

$$\mathbf{a}b\ \mathbf{c}d$$

where OPT faults only on a and c. Since OPT does not fault on d, that page must be in OPT's cache at that point. By definition of FIFO, d cannot be b or c, since those pages are still in FIFO's cache when d is requested. If d is also not a, then after OPT loads a, both a and d must be in the cache (we may assume that OPT only faults on a page if it is requested). But then OPT would fault on b, since $b \neq a$. Hence d is a. Since a remains in the cache when OPT serves c, either a or c must be loaded into the expensive cache by OPT.

Therefore OPT has at least one expensive fault per subinterval. □

Lemma 3.3 *For any relevant request sequence σ of length ℓ, we have*

$$\text{OPT}(\sigma) \geq \min\left(\frac{2}{3}, \frac{1+x}{4}\right) \cdot \ell.$$

Proof. We partition σ into relevant intervals. Consider any relevant interval I and denote its length by $\ell(I)$. We show that for each such interval, $\text{OPT}(I) \geq \min(2/3, (1+x)/4)\ell(I)$.

If $\ell(I) = 1$, $\text{OPT}(I) \geq 1 \geq 2/3$. If $\ell(I) = 3$, $\text{OPT}(I) \geq 2 \geq \frac{2}{3} \cdot 3$.

Suppose $\ell(I) \geq 5$, and write $f = \lfloor \ell(I)/4 \rfloor$. We use Lemma 3.2.

If $\ell(I) = 4f + 1$ then OPT has $2f + 1$ faults, among which at least f are expensive. This is a cost of $(f + 1 + fx)/(4f + 1)$ per request.

If $\ell(I) = 4f + 3$ then OPT has $2f + 2$ faults, among which at least f are expensive. This is a cost of $(f + 2 + fx)/(4f + 3)$ per request.

Both of these are at least $(1 + x)/4$ or at least $2/3$. □

We are now ready to prove Theorem 3.1.

Proof of Theorem 3.1. Consider a relevant request sequence σ. We partition σ into marking phases, and compare the cost of FIFO to the *average* cost of OPT per marking phase. By Lemma 3.3, we have that $\text{OPT}(p) \geq \min(4/3, (1 + x)/2)$ for any marking phase p on average. We also have $\text{FIFO}(p) = 1 + x$ exactly. Therefore, $\mathcal{R}(\text{FIFO}) \leq \max(2, \frac{3}{4}(1 + x))$.

To show that this is the exact ratio, we use a more general lower bound on the performance of FIFO, that holds if both the cheap cache and the expensive cache have some size $k \geq 1$. Consider the following request sequence:

$$(1\ 2\ 3\ \ldots\ 2k + 1)^N.$$

We define two different offline strategies to serve this sequence, and use one or the other depending on x.

For $x \geq \frac{2k+3}{2k+1}$, OPT has $1, \ldots, k$ in the cheap cache at the start, and $k+1, \ldots, 2k$ in the expensive cache. OPT only faults on pages $1, \ldots, k$ and $2k+1$, each time evicting the page from this set which will be requested the furthest in the future (LFD [4]). In $2k+1$ marking phases, there are $2k(2k+1)$ requests in total and $2k(k+1)$ requests to pages $1, \ldots, k$ and $2k+1$. OPT has a fault once every k requests to these pages, so it has $2(k+1)$ faults in $2k+1$ phases.

FIFO pays $k(1+x)$ per phase, so by letting N grow without bound we find

$$\mathcal{R}(\text{FIFO}) \geq \frac{2k+1}{2k+2}k(1+x)\ .$$

For $x < \frac{2k+3}{2k+1}$, OPT uses its entire cache in a round-robin fashion. Thus OPT pays on average $(1+x)/2$ per FIFO phase, so

$$\mathcal{R}(\text{FIFO}) \geq 2k.$$

This matches the upper bounds for $k=1$. □

FIFO cannot have optimal competitive ratio for every x since according to [6] there exists a constant competitive algorithm whereas the competitive ratio of FIFO grows linearly with x. However for relatively small values of x, FIFO has optimal competitive ratio .

Lemma 3.4 FIFO *is an optimal on-line algorithm for the related paging problem and* $k=1$ *in the interval* $1 \leq x \leq \sqrt{249}/6 - 1/2 \approx 2.1299556$.

Proof. The upper bound follows from the analysis of FIFO. We prove the lower bound.

We construct a sequence of requests, which consists of requests for at most three different pages: 0, 1 and 2. Consequently there are 6 different possible configurations of the cache. Two basic ingredients for the proof are similar to the lower bound for the k-server problem [9]. Many new ingredients are added to the analysis, that becomes more involved for the related case.

As in [9], the sequence of requests is constructed in such a way that the on-line algorithm has a fault in every request. That is, in each step a request is added for the only page that is present in none of the two caches of the on-line algorithm. In order to give a lower bound on the general (not strict) competitive ratio, we build a long enough sequence. Note that the on-line cost of such a sequence of N requests is at least N. Another similarity to [9] is a comparison of the on-line algorithm to several off-line algorithms. We keep five off-line algorithms that all process the complete sequence along with the on-line algorithm, and consider their average cost.

A sequence of requests that is constructed in this way can be transformed into a sequence of costs that the on-line algorithm pays in every step. We define a cost sequence as a sequence of 1's and x's, where 1 indicates a fault of the cheap cache whereas x indicates a fault of the expensive cache. Given a starting configuration of the on-line algorithm (without loss of generality $C = \{0\}$ and $E = \{1\}$) and a cost sequence, it is easy to recall the request sequence. We define a set of pattern sequences S which consists of a finite number of cost sequences of bounded length s. The patterns in S form a prefix code, or in other words, there is a unique way to break any cost sequence into members of S (a remainder of length at most s might be left over). We call the points in the cost sequence where a member of S ends "breakpoints". The starting point of the complete sequence is also considered a breakpoint.

Besides having to serve the request sequence, a second requirement from the off-line algorithms is that at every breakpoint, each of the six algorithms (that is, the five off-line algorithms and the on-line algorithm) has a different configuration. This means that at every breakpoint, exactly one algorithm has each possible configuration. Note that we do not require a certain order among the configurations of the off-line algorithms.

For each of the patterns $p \in S$, we compute the cost of the on-line algorithm on p and compute the cost of the five off-line algorithms to serve all requests and end up in a valid configuration (i.e. so that all six final configurations are again all different). Note that this gives room to the off-line algorithms to choose which configuration each of them ends up in, and there are $5! = 120$ possibilities.

As we need to show a lower bound of $\mathcal{R}$, we compare the on-line cost on p, denoted by $\text{ONL}(p)$, to the total cost of the off-line algorithms $\text{OFF}(p)$, and show $5 \cdot \text{ONL}(p) \geq \mathcal{R} \cdot \text{OFF}(p)$. This implies that the total off-line cost is at most

$$\text{OFF}(\sigma) \leq \frac{5}{\mathcal{R}} \cdot \text{ONL}(\sigma) + 5(1+x) + 5(s-1)x.$$

The value $5(1+x)$ is the set-up cost of the five off-line algorithms to reach the correct starting configurations, and $5(s-1)x$ is an upper bound on the cost of serving the "remainder" of the request sequence which is not a pattern in S. As OPT is an optimal off-line algorithm, its cost is at most the average cost of the five off-line algorithms and so $\text{OPT}(\sigma) \leq 1/\mathcal{R} \cdot \text{ONL}(\sigma) + 1 + sx$. We get that

$$\text{ONL}(\sigma) \geq \mathcal{R} \cdot \text{OPT}(\sigma) - 1 - sx.$$

It is left to show the sets S that imply the lower bound. We show a pattern set S_1 that proves a lower bound of 2 for every value of x in the interval $[1, 5/3]$, and a pattern set S_2 which proves the lower bound $3(1+x)/4$ for every value of x in the interval $[5/3, 2.1299556]$. See Table 1. To check that those patterns are indeed valid (give the correct lower bound on the competitive ratio) it is only required to solve a linear or a quadratic equation for each pattern p. It is also easy to see that each pattern set forms a prefix code. The cost of the on-line algorithm follows directly from the pattern whereas the off-line costs require a short proof. The details of that are omitted. □

Even though all the data was carefully verified, the origin of the pattern sets is a computer program we used. The program performs an exhaustive search on all 120 permutations and finds the cheapest cost of the off-line algorithms for each possible pattern. The result given here is an output of the program when it checked all patterns up to a length of 11. Using this program for patterns of length 13, we also found that FIFO is optimal for $x \leq 2.206$. An extension of the program for patterns of length at most 15 improves the bound by a negligible amount.

Interestingly, it is possible to show that LRU is strictly worse than FIFO. We have the following result.

Lemma 3.5 $\mathcal{R}(\text{LRU}) = 1 + x$.

Proof. The upper bound follows from Theorem 2.1.

Denote the starting caches of LRU by $C = \{0\}$ and $E = \{1\}$. Without loss of generality, we may assume LRU starts by using the cheap cache. We use the following sequence: $(202101)^N$. For one iteration of this sequence (six requests), LRU pays $2 + 2x$. On the other hand, an offline

Pattern p	$\text{ONL}(p)$	$\text{OFF}(p)$
111	3	7
$xx1$	$2x+1$	$2x+5$
$11x$	$x+2$	$x+6$
xxx	$3x$	$2x+5$
$x111$	$x+3$	$x+9$
$x1x1$	$2x+2$	$5x+5$
$1x1x$	$2x+2$	$5x+5$
$1xx1$	$2x+2$	$2x+8$
$x11x$	$2x+2$	$2x+8$
$x1xx$	$3x+1$	$4x+6$
$1xxx$	$3x+1$	$2x+8$
$1x111$	$x+4$	$3x+8$
$1x11x11$	$2x+5$	$4x+13$
$1x11xx1$	$3x+4$	$4x+13$
$1x11x1x$	$3x+4$	$5x+12$
$1x11xxx$	$4x+3$	$3x+14$

Pattern p	$\text{ONL}(p)$	$\text{OFF}(p)$
x	x	$x+2$
$11x$	$x+2$	$x+6$
$1xx$	$2x+1$	9
1111	4	8
$111x$	$x+3$	$x+7$
$1x11x$	$2x+3$	15
$1x1xx$	$3x+2$	$x+14$
$1x1111$	$x+5$	$x+13$
$1x111x$	$2x+4$	$2x+12$
$1x1x1x$	$3x+3$	20
$1x1x11x$	$3x+4$	$x+20$
$1x1x1111$	$2x+6$	$5x+11$
$1x1x111x$	$3x+5$	$3x+17$

Table 1: Left are the patterns for $x \in [1, 5/3]$, right for $x \in [5/3, 2.1299556]$.

algorithm can keep 0 in E at all times and only pay 2 per iteration. By letting N grow without bound, we get the desired result. $\square$

We end this section with a short analysis of two other natural algorithms. When x is large, it is tempting to use an algorithm which uses only the slow cache, i.e. each time there is a page fault, the algorithm replaces the page in C by the requested page. This algorithm is not competitive at all. Consider the on-line cache in the beginning of the sequence, and denote $C = \{a\}$ and $E = \{b\}$. Let c and d be pages such that all four pages a, b, c, d are distinct. The sequence simply alternates between pages c and d. Let $2N$ be the length of the sequence. Then the cost of the algorithm is $2N$, whereas OPT can initialize its caches by $C = \{c\}$ and $E = \{d\}$ and pay only $1 + x$. As N grows, the competitive ratio grows without bound.

Another natural option is to apply the BALANCE algorithm, trying to balance the costs of the two caches. As BALANCE performs as well as LRU and FIFO for $x = 1$ (competitive ratio 2 for paging), it is interesting to see how it performs for other values of x. The modified definition of BALANCE for $x > 1$ is as follows:

Keep a counter for each place in the cache (B_C and B_E). On a page fault, if $B_C + 1 \leq B_E + x$, replace the page in C and increase B_C by 1; otherwise replace the page in E and increase B_E by x.

Lemma 3.6 *The competitive ratio of* BALANCE *is at most* $2x$*. This is tight for infinitely many values of* x*.*

Proof. We again reduce a request sequence σ to a relevant request sequence and cut it into parts. Each part contains some number of cheap faults followed by one expensive fault, except for the last part which might not contain an expensive fault. Let N be the number of parts that contain an expensive fault. The cost of BALANCE consists of two parts; cost for cheap faults and cost for expensive faults. These are stored in B_C and B_E. It is easy to show that after a page is put into E (i.e. at the start of a new part), we have $B_C \leq B_E < B_C + 1$. Hence the cost of BALANCE is $B_E + B_C > Nx + (Nx - 1) = 2Nx - 1$. To give a bound on OPT, note that there are $N - 1$ subsequences where BALANCE has a cheap fault followed (immediately) by an expensive fault followed by a cheap fault (of the next part). These 3 faults must be on 3 distinct pages and hence OPT must have at least one fault, or $N - 1$ faults in total. Consequently, the competitive ratio approaches $2x$ as the length of the sequence grows.

To show tightness we consider (integer) even values of x, say $x = 2y, y \in \mathbb{N}$. Let $C = \{0\}$ and $E = \{1\}$ the initial configuration of BALANCE. The sequence consists of N parts, each of which is the sequence $(20)^y 2(10)^y 1$. BALANCE has a fault on every request and pays $4x$ for each part. However, an offline algorithm kan keep 0 in E at all times and only have two faults per part. This gives the competitive ratio $2x$. □

4 Lower bounds for trackless algorithms

In this part we prove a lower bound for trackless algorithms which is linear in kx. First we prove the lower bound for the case $k = 1$, and then for general k.

Theorem 4.1 *For $x \geq 2$, any trackless algorithm for $k = 1$ has a competitive ratio of at least $x/2 + 1$.*

Note that for $x < 2$, we can use the general lower bounds of the previous section. Specifically Lemma 3.4 gives a lower bound of 2 for the complete interval $1 \leq x \leq 2$.

Proof. We construct a sequence in pieces, and bound the ratio for the pieces. We have a distinguished page denoted by a, this page is kept in E by OFF, and is never placed into E by ONL. The configurations of the caches are called inverse if the caches of OFF are in the configuration $E = \{a\}$, $C = \{y\}$ and the caches of ONL are in $E = \{y\}$, $C = \{a\}$ for some page y. Each piece starts and ends when the algorithms are in an inverse configuration.

The first inverse configuration can be reached as follows. Denote the page that ONL has in cache C by a and the page that it has in E by b. Then, if this is not already the case, OFF loads a into E and b into C. Thus OFF has a startup-cost of at most $1 + x$. We are now ready to start the first piece.

Suppose the caches are ONL : $C = \{a\}, E = \{b\}$; OFF : $C = \{b\}, E = \{a\}$. The next request is for c. We use a case analysis.

Case 1 c is placed by ONL into E. OFF loads c into C and the piece ends. The configuration is inverse and the cost ratio during this piece is $x \geq x/2 + 1$ for $x \geq 2$.

Case 2 c is placed by ONL into C.

Case 2.1 The next page is placed by ONL into E. Then this request is for the page d. Denote by ℓ_E the number of pages which are placed into E by ONL after this request. These ℓ_E requests are alternating between b and d, and the $\ell_E + 1$-st request is for a. (If ONL never puts another page into C, then this is the last piece. By extending it arbitrarily long, the ratio for this piece tends to $x \geq x/2 + 1$.) After the $\ell_E + 1$-st request we are again in an inverse configuration. During this piece the cost of OFF is $\ell_E + 2$, and the cost of ONL is $(\ell_E + 1)x + 2$. Therefore the ratio is at least $x/2 + 1$.

Case 2.2 The next page is placed by ONL into C. Then this is a request for a.

Case 2.2.1 The next page is placed into E. Then we request d, and the piece is ended. OFF has cost 2, and ONL has a cost of $x + 2$.

Case 2.2.2 The next page is placed into C. Then we request c, and we are at the same configuration as we were after the first request for c (the start of case 2). During this loop OFF has cost 0, and ONL has cost 2. We can therefore ignore this loop and continue as at the start of case 2. If ONL never puts another page in E, then this is the last piece. By extending it arbitrarily long, the ratio for this piece tends to ∞.

Since we considered all the possible cases, and the ratio of the costs is at least $x/2 + 1$ in all cases, we are done. □

Theorem 4.2 *Any trackless algorithm for $k \geq 2$ has a competitive ratio of at least $c \geq k(x+1)/2$.*

Proof. Consider an arbitrary trackless online algorithm. Consider two sets of pages, $A = \{a_1, \ldots, a_{k+1}\}$ and $B = \{b_1, \ldots, b_{k+1}\}$. The sequence is constructed in such a way that the online algorithm always has pages from A in C, and pages from B in E. If the algorithm will place the next page into C, then this request is to the page from A which is not currently in C. Otherwise, the request is to the page from B which is not currently in E.

Consider a long sequence of requests produced by this rule, denote by p the number of online faults in C, and by q the number of faults in E. Then ONL has a cost of $p + qx$.

To estimate the offline cost for the request sequence, we consider two offline algorithms. The first algorithm OFF_1 uses E for $A_1, \ldots, A_k$ and one memory cell from C for A_{k+1}. Furthermore it uses the other $k-1$ cells (called active cells) for serving the requests for B, always evicting the page from B which will be requested the furthest in the future (LFD [4]). The other offline algorithm called OFF_2 works in the same way, but with the roles of A and B interchanged. Thus OFF_2 has the pages from A continuously in its cache.

Consider first the cost of OFF_1. It has at most $kx + 1$ cost on the requests from the set A, and a starting cost of $k - 1$, placing the first $k - 1$ requests for the pages from set B. We can bound the remaining cost in a similar way as it is done in [11]. We partition the sequence of requests to pages in B into subsequences of length k called k-parts. Consider a k-part. Suppose there is a request for a page from B, which is not contained in the $k - 1$ active cells of OFF_1. Consider the first such request in the k-part. Denote the set of active cells after servicing this request by C', the evicted page by b_i, and the remaining element of $B \setminus (C' \cup \{b_i\})$ by b_j. If during the rest of this k-part there is a request for b_j, then there will be no other fault in the k-part: by the LFD rule there is no further request for b_i or for the page which was evicted when b_j was placed into the cache. If there is no request for b_j, then there is no fault during the next $k - 2$ requests (there is no request for b_i).

This yields that OFF_1 has at most 2 faults during a k-part. Therefore we showed that the cost of OFF_1 is at most $k(x+1) + \lceil 2q/k \rceil$. Similarly, OFF_2 has a cost of at most $k(x+1) + \lceil 2p/k \rceil$. Therefore, we obtained that the optimal offline cost is at most $(k+1)x + \lceil 2\min(p,q)/k \rceil$. By considering arbitrarily long sequences, where we have $p + q \to \infty$, we find that the competitive ratio of any on-line algorithm is at least

$$\frac{p + qx}{2\min(p,q)/k} \geq \frac{(1+x)k}{2},$$

which ends the proof. $\square$

Corollary 4.1 *Any trackless algorithm has a competitive ratio at least half that of any marking algorithm.*

References

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th ACM Symposium on the Theory of Computing*, pages 305–313. ACM, 1987.

[2] W. W. Bein, R. Fleischer, L. L. Larmore, Limited bookmark randomized online algorithms for the paging problem, *Information Processing letters*, 76: 155–162, 2000.

[3] W. W. Bein, L. L. Larmore, Trackless online algorithms for the server problem, *Information Processing letters*, 74 no 1-2: 73–79, 2000.

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

[5] Marek Chrobak and John Noga. Competitive algorithms for multilevel caching and relaxed list update (extended abstract). In *Proceedings of the Ninth ACM-SIAM Symp. on Discrete Algorithms*, pages 87–96. ACM/SIAM, 1998.

[6] M. Chrobak, J. Sgall, The weighted 2-server Problem, in *Proc. of STACS 2000, LNCS 1770*, Springer-Verlag Berlin, 593–604, 2000.

[7] A. Fiat, M. Ricklin, Competitive algorithms for the weighted server problem, *Theoretical Computer Science*, 130: 85–99, 1994.

[8] A. Karlin, M. Manasse, L. Rudolph, D. Sleator, Competitive snoopy caching, *Algorithmica*, 3(1): 79–119, 1988.

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

[10] L. Newberg, The K-server problem with distinguishable servers, Master's Thesis, Univ. of California at Berkeley, 1991.

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

[12] N. Young, Online file caching, *Proc. 9th Annual ACM-SIAM Symp. on Discrete Algorithms*, 82–86, 1998 (to appear in *Algorithmica*)