Paging with Request Sets*

Leah Epstein† Rob van Stee‡ Tami Tamir§

August 31, 2006

Abstract

A generalized paging problem is considered. Each request is expressed as a set of u pages. In order to satisfy the request, at least one of these pages must be in the cache. Therefore, on a page fault, the algorithm must load into the cache at least one page out of the u pages given in the request. The problem arises in systems in which requests can be serviced by various utilities (e.g., a request for a data that lies in various web-pages) and a single utility can service many requests (e.g., a web-page containing various data). The server has the freedom to select the utility that will service the next request and hopefully additional requests in the future.

The case $u = 1$ is simply the classical paging problem, which is known to be polynomially solvable. We show that for any $u > 1$ the offline problem is NP-hard and hard to approximate if the cache size k is part of the input, but solvable in polynomial time for constant values of k. We consider mainly online algorithms, and design competitive algorithms for arbitrary values of k, u. We study in more detail the cases where u and k are small. We also give an algorithm which uses resource augmentation and which is asymptotically optimal for $u = 2$.

1 Introduction

Modern operating systems have multiple memory levels. In simple structures, a memory is organized in equally sized pages. The basic paging model is defined as follows. The system has a slow but large memory (e.g. disk) where all pages are stored. The second level is a small, fast memory (cache) where the system brings a page in order to use it. If a page which is not in the faster memory level is requested, a page fault occurs, and a page must be evicted in order to make room to bring in the new page. The processor must slow down until the page is brought into memory, and in practice, for many applications, the performance of the system depends almost entirely on the number of page caching (uploads to the cache). We define the cost of an algorithm as simply the total number of page uploads.

Traditional paging problems assume that at every step of a request sequence, there exists a unique page that can fulfill the needs of the system. This page must be loaded into the cache if it does not reside there already. Such a situation is plausible, however, often the need for a very

*A preliminary version of this paper appeared in *Proc. Scandinavian Workshop on Algorithm Theory (SWAT 2006)*, p. 124-135.

†Department of Mathematics, University of Haifa, 31905 Haifa, Israel. lea@math.haifa.ac.il.

‡Department of Computer Science, University of Karlsruhe, D-76128 Karlsruhe, Germany. vanstee@ira.uka.de.
Research supported by Alexander von Humboldt Foundation.
§School of Computer Science, The Interdisciplinary Center, Herzliya, Israel. tami@idc.ac.il.

specific page is not acute and the need is for a certain piece of *information* rather than a certain page.

For instance, on the *world wide web*, information is often mirrored across many websites (e.g. currency exchange rates). In such a situation, it makes sense to make a list of several places where the information can be found, and allow the system to conveniently choose from them. Another application is a *media broadcasting system*. In such a system, media files, which are the smallest media units that can be loaded into the system, are kept on disks. Media files are replicated and each is stored on several, not necessarily uniform, disks. The content of each disk is known. A broadcast is defined by a list of required media in some specific order (for example, list of songs to be transmitted). The goal is to broadcast all media while minimizing the number of disks loadings.

Let U_j denote the j-th request, that is, U_j is the set of pages containing the information required in the j-th request. In order to keep the running times of paging algorithms low, we fix the number of options given to the algorithm in every request to be a parameter u. Formally, for all j, we assume that $|U_j| = u$. The size of the cache is denoted by k. The request sequence is denoted by σ, and n denotes the total number of different pages that occur in σ. Given a set of requests $\{U_1, \ldots, U_j\}$, we say that a set S *covers* this set of requests if for all $i, 1 \leq i \leq j$, it holds that $S \cap U_i \neq \emptyset$. In online paging problems, each element in the request sequence arrives after the previous request was serviced (i.e. the decision on the eviction of another page was made). The competitive ratio is the asymptotic worst case ratio between the cost of an online algorithm and the cost of an optimal offline algorithm OPTwhich knows all requests in advance.

Related Work For the classical offline problem, there exists a polynomial simple optimal algorithm LFD (Longest Forward Distance) designed by Belady [2]. LFD always evicts the page for which the time until the next request to it is maximal. Two common paging algorithms for the classical paging problem are LRU (Least Recently Used) and FIFO (First In First Out). LRU computes for each page which is present in the cache the last time it was used and evicts the page for which this time is minimum. FIFO acts like a queue, evicting the page that has spent the longest time in the cache. Variants of both are common in real systems. Although LRU outperforms FIFO in practice, LRU and FIFO are known to have the same competitive ratio of k. Further this ratio is known to be the best possible, see [14, 10]. Randomized algorithms were studied by [7, 1, 12]. See [9] for a survey on online paging problems.

One generalization of paging was studied by Fiat and Ricklin [8]. They studied the weighted paging problem (i.e., where each slot in the cache may have a distinct cost of replacing the page stored in it), and gave algorithms with a doubly exponential upper bound in k. They showed that the competitive ratio of any algorithm for this problem is at least $k^{\Omega(k)}$. For the special case where only two weights are allowed they have a $k^{O(k)}$-competitive algorithm.

As explained in Section 2, paging with request sets captures other well-studied problems such as set cover and hypergraph vertex cover [6]. The online problem captures dynamic versions of the above problems such as online vertex cover. However, in the studied version of online vertex cover (see e.g., [5]), the input graph is revealed vertex by vertex, while our problem induces a problem in which the graph is revealed edge after edge. Both our results and the results in [5] imply that the online problem is significantly harder than the offline one.

The online version of our problem is a special case of metrical task systems and specifically of their sub-class, *metrical service systems* (also called *forcing tasks systems*). For details, see [3, 11, 4].

Our Results We show that unlike the classical paging problem, paging with request sets is NP-hard and in fact hard to approximate within a factor of $\Omega(u)$ unless P = NP. If k is fixed, then the problem can be solved via dynamic programming.

We further study the online problem. We show that natural extensions of LRU and FIFO are not competitive. We present competitive algorithms for all values of u and k. We consider the paging model described above as well as the same paging model with bypassing. Note that even though the competitive ratios of our algorithms are quite high for most variants (i.e. exponential in k), we show that in many of the cases this is unavoidable. This is similar to the generalization of paging considered in [8] that also results in high competitive ratios.

Finally, we present a simple online algorithm which uses resource augmentation. Here the offline algorithm that it is compared to is restricted to a cache size of $h < k$. This generalization was considered already by Sleator and Tarjan in [14]. We show that the competitive ratio of our algorithm tends to the optimal value of 2 for $u = 2$ and large values of k/h.

2 The Offline Problem

We begin by describing a dynamic program (DP) for the problem. For a set S of size exactly k, let $P_{j,S}$ denote the cost of servicing the first j requests of σ in a way that the cache content after the jth request is S. Since the jth request is the last to be serviced, $P_{j,S}$ is defined only for $S \cap U_j \neq \emptyset$ (or defined to be ∞ when this does not hold). An optimal solution can be obtained using the following dynamic program. Here we assume the optimal total cost is at least k.[1]

Initialization For $j \leq k$, set $P_{j,S} = k$ if S covers $U_1, \ldots, U_j$. Else, set $P_{j,S} = \infty$.

In the initialization, we consider all possible ways to fill the cache. This requires k loads and therefore has the cost k. We assume that no page replacements are done as long as the cache has space to load pages, thus, we ignore these sets that do not cover the first j requests (by setting their price to ∞).

Step For $j > k$, if $S \cap U_j = \emptyset$, set $P_{j,S} = \infty$. Else set

$$P_{j,S} = \min\left\{P_{j-1,S},\ 1 + \min\{P_{j-1,S'} | S = S' \cup \{x\} \setminus \{y\} \text{ for some } x \in U_j\}\right\}.$$

For $j > k$, when calculating $P_{j,S}$ for $S \cap U_j \neq \emptyset$ there are two cases. The first is when no upload is done for servicing the jth request, that is, the current content of the cache covers U_j. In this case the total cost of servicing the first j requests is equal to the cost of servicing the first $j-1$ requests. The second case is when an upload is essential. If the content of the cache, S', does not cover U_j then some page $y \in S'$ is removed and a page $x \in U_j$ is inserted. The cost in this case is one for the current upload plus the cost of servicing the first $j-1$ requests in a way that the cache content before the jth request is S'. Therefore, the minimal cost of servicing the first j requests is determined by the optimal S'. To calculate $P_{j,S}$, the minimum cost out of these two cases is considered.

The optimal cost for the whole sequence is the minimal value of $P_{|\sigma|,S}$ for some set S. The size of the DP table is polynomial in n: there are n^k possible sets S, for each such set the value $P_{j,S}$ is calculated for $|\sigma|$ different values of j. Each entry is calculated in time $O(uk)$.

[1] This assumption can be removed by considering also sets of size smaller than k. We skip this technical extension.

Corollary 2.1 *Paging with request sets can be solved in time* $O(|\sigma|ukn^k)$.

Clearly, the above DP algorithm is polynomial only for constant k. While the offline traditional paging problem is known to be optimally solved for arbitrary k, this is not the case for the generalized problem. In particular, we show that our problem is NP-hard even for request sets of size 2.

Theorem 2.2 *Offline paging with request sets and an arbitrary cache size* k *is NP-hard even for* $u = 2$.

Proof: Reduction from *Vertex Cover* (VC). Given an instance for VC, $G = (V, E)$, and the question whether G has a vertex cover of size k, construct the following instance for paging with request sets. The sequence σ consists of $|E|$ requests; the jth request is $U_j = \{v_{j1}, v_{j2}\}$, where (v_{j1}, v_{j2}) is the jth edge (in arbitrary order) of E. It is easy to verify that it is possible to service all the requests at a total cost of k if and only if G has a vertex cover of size k. □

This reduction can be generalized to show that for arbitrary sizes of sets the problem is as hard as set-cover. Thus, it cannot be approximated within factor $\Omega(\log n)$ [13]. The reduction from vertex cover can be extended for any instance with uniform size request sets, that is, when $|U_j| = u$ for all u.

Theorem 2.3 *Assuming* $P \neq NP$, *the optimal cost of paging with request sets for instances having request set of size* u *cannot be approximated in polynomial time within factor* $(u - 1 - \varepsilon)$.

Proof: We show an L-reduction from hypergraph vertex cover, for which this hardness result is known [6]. Let S be an instance of hypergraph vertex cover (HVC) with k nodes, and let *opt* be a minimal size vertex cover of S. Build an instance σ_s for our paging problem by listing the hyperedges of S in some order.

Consider a cache of size k. An optimal algorithm can service σ_s at cost $|opt|$ by placing the nodes of the vertex cover in the cache. Consider any algorithm that services σ_s. The set of vertices that are inserted into the cache along the whole sequence form an HVC. The cache size is selected to be k so no deletions are required. Therefore, the cost of servicing σ_s is the size of the HVC found by the algorithm. □

3 The Online Problem

3.1 The Performance of Standard Algorithms

In this section we show that several reasonable versions of LRU and FIFO, adapted for paging with request sets, are not competitive. To generalize the algorithms, we need to define the behavior on a page fault. Specifically, we not only need to define the method of page eviction but also the method of choosing a page of the new request to be inserted to the cache.

The page eviction method of FIFO is identical to the original algorithm, that is, remove from the cache the page that was inserted first. For LRU, we say that a page is *used* if it appears in a request (but not necessarily downloaded), thus, LRU removes from the cache a page that appeared least recently in a request. Ties are broken arbitrarily. We mention that our non-competitiveness proof below is suitable also for other eviction methods of LRU, like removing the page that was least recently "essential", that is, loaded or serviced a request.

For analyzing the loading page method, we first consider a situation where the choice of a new page is arbitrary. In particular, it might be that the page inserted to the cache is the one that has been out of the cache the longest time (i.e., has never been in the cache, or has been evicted least recently). The same example is applicable to both LRU and FIFO.

Lemma 3.1 *The above versions of* LRU *and* FIFO *are not competitive for paging with request sets.*

Proof: Given $k \geq 2$, and $u \geq 2$, let $\{a_0, \ldots, a_{u+k-1}\}$ denote $u+k$ designated pages. For convenience of notation, define a_j for $j \geq u+k$ to be $a_{j \mod (u+k)}$.

The sequence of requests repeats the following subsequence $\sigma_0, \ldots, \sigma_{u+k-1}$. Request σ_i is defined to be $\{a_i, \ldots, a_{i+u-1}\}$. Both LRU and FIFO have the cache contents $\{a_{i+u}, \ldots, a_{i+u+k-1}\}$ before this request, where pages are listed in the order in which they will be evicted. The page that has been out of the cache for the longest time is $a_i = a_{i+u+k}$. Clearly each request is a fault.

However, OPT keeps in its cache the pages $a_{u \cdot \ell}$ for all $\ell \geq 0$ such that $u \cdot \ell < u+k$. The number of such pages is $\lceil \frac{k}{u} \rceil + 1$. Since $u \geq 2$, it is clear that this number never exceeds k. Thus, after loading these pages in its cache, OPT never makes another fault. This proves the lemma. □ Next, consider versions of LRU and FIFO that prefer to insert into the cache a page that was evicted most recently. In this case we use a set of $k+1$ pages $\{a_0, \ldots, a_k\}$, and $u-1$ additional pages $\{b_1, \ldots, b_{u-1}\}$. Similar to the previous example, we let a_j for $j \geq k+1$ to be $a_{j \bmod (k+1)}$. The sequence repeats the sub-sequence of requests $\tau_0, \ldots, \tau_k$ where $\tau_i = \{a_i, b_1, \ldots, b_{u-1}\}$. The requests b_j are never inserted into the cache. Before τ_i arrives, the cache contains pages $\{a_{i+1}, a_{i+2}, \ldots, a_{i-1}\}$ (listed in the order in which they are to be evicted). LRU and FIFO fault on every request, whereas OPT keeps the pages $\{b_1, a_1, \ldots, a_{k-1}\}$ in its cache. We may conclude that LRU and FIFO are not competitive.

Since the standard algorithms fail, in the next subsections we design very different algorithms for our problem. These algorithms try to track the configuration of OPT in order to remain competitive.

3.2 A Competitive Algorithm

Our algorithm works in phases. A phase ends when it must be the case that OPT had a fault. Consider a single phase. Let C be a collection of sets $S_1, S_2, \ldots$ each of size at most k (cache size) such that each set S_i covers the requests presented so far in the phase. If C is empty then OPT must miss and a new phase begins. Otherwise try to make the cache of the online algorithm ONL as similar to the sets of C as possible. Specifically, ONL knows the set C. In every step (miss) ONL tries to exclude a set from C.

We use the following assumptions. Algorithms silently ignore requests which do not cause faults. In the analysis we may assume that each request is indeed a fault: requests which are not faults can only increase the optimal cost. Thus we simply remove non-fault requests from the request sequence before starting our analysis. Note that our algorithms may replace more than one page on a fault.

Our construction will lead to the following general theorem.

Theorem 3.2 *For paging with request sets of size u and a cache of size k, there exists an algorithm* $\text{ALG}_u(k)$ *which has a competitive ratio of* $\frac{u^{k+1}-u}{u-1}$ *. Moreover, for any constant k, the competitive ratio of any online algorithm is* $\Omega(u^k)$.

We first describe a competitive online algorithm for the case $u = k = 2$. Below, we will show how to use this algorithm as a subroutine in more general cases.

The algorithm ALG works in phases. A phase is a subsequence of requests, where it can be proved that OPT has made a fault either on one of the requests of this phase (excluding the very first one) or on the first page of the next phase. In the sequel, we analyze the contents of the cache of OPT in the case that it did not make a fault in the current phase (except, possibly, on the first request). We denote the (pages of the) first request of a phase by $\{a, b\}$. Our algorithm will insert b in the cache. Throughout the phase, ALG always keeps one page out of $\{a, b\}$ in its cache. The situation of OPT is similar if it did not have a fault yet.

The phase continues until we know that OPT has a fault, or will have a fault on the current request. This request will start a new phase. We assume that OPT does not make a fault, until this leads to a contradiction. The easiest and most important case is the case where three independent (non-overlapping) sets are requested. This clearly implies a fault of OPT, and the request for the third independent set starts a new phase. There are two cases for the second request.

Case 1: Request 2 is for an independent set. Denote request 2 by $\{c, d\}$. The algorithm loads c into the cache and has $\{b, c\}$ in the cache. From now on, ALG also keeps one page from $\{c, d\}$ in its cache during this phase. Thus, the remaining requests are serviced by loading a matching page (if possible). Since we assume that OPT does not make a fault, OPT now has one page from $\{a, b\}$ and one page from $\{c, d\}$ in its cache.

If a new independent set is requested ($\{e, f\}$), then OPT must have a fault and this request starts a new phase. As long as this does not happen, requests always have exactly one page in common with at least one of the first two requested pairs. The algorithm always loads such a page. Thus in this phase, it never loads a page outside of the set $\{a, b, c, d\}$. We analyze the options for the next requests.

Case 1A If request 3 consists of one page from $\{a, b, c, d\}$ and one other page, then OPT must have that page from $\{a, b, c, d\}$ in its cache or make a fault. When this happens, ALG fixes this page in its cache and does not evict it anymore. Thus it mimics OPT in the case that OPT has not made a fault yet.

Suppose a is fixed (as a result of request $\{a, e\}$). Then a is not requested anymore in this phase (because ALG keeps it in its cache). We have two cases for request 4. If b is requested with an outside page, we have a third independent set unless this page is e. However, in this particular case OPT must have two pages out of $\{a, b, e\}$ to cover them and one page from $\{c, d\}$ which is not possible. So a new phase starts with this request (phase length is 3).

The other case is when request 4 contains c or d, together with b or an outside page. In this case, this page (c or d) is fixed in the cache. The entire cache is now fixed and therefore request 5 can start a new phase. Consequently, if Case A occurs, we defined a phase which consisted of at most four requests.

Case 1B If request 3 consists of one page from $\{a, b\}$ and one page from $\{c, d\}$, then ALG has a choice which page to evict. In this case it initially evicts an arbitrary page, but keeps track of these type of requests, which are called *bridges*. If two bridges overlap in a page, say a, then OPT must either have a or make a fault – since it cannot have all of b, c and d in its cache. In such a case ALG fixes that page in its cache. It can be seen that if two bridges are requested in sequence, they must overlap in a page. From the first bridge, ALG only loads one page and so its state does not match

the bridge. This means that the only non-overlapping bridge cannot be requested immediately afterwards.

Consider request 4. If it contains one request to an outside page, this brings us back to case 1, and we get a phase which consists of at most five requests. Otherwise, there are two bridges in succession, and therefore again ALG has one page fixed. This means that after at most four requests in a phase, at least one page in the cache is fixed, and the next request fixes the other page. Thus the maximum length of a phase is 5. ALG makes at most five uploads per at least one upload of OPT and therefore it is at most 5-competitive.

Case 2: Request 2 is not independent. Denote it by $\{a, c\}$. The proof goes along the lines of the previous case. The algorithm will insert a into the cache and will thus have $\{a, b\}$. If OPT does not fail on the second request, it either has the cache contents $\{b, c\}$ or it has a in its cache. Request 3 must be to a set independent of $\{a, b\}$. It can contain two new pages, $\{d, e\}$ or c with a new page d.

Case 2A If request 3 is for two new pages, and OPT still has not failed, then the contents of the cache of OPT is a with one of these pages. ALG moves d into its cache and evicts b. There can be one additional request in the current phase on which OPT does not fail, and that is a request for e with an additional page. This fixes the cache of ALG to $\{a, e\}$ and the fifth request starts a new phase. We got a phase of length four.

Case 2B If request 3 is for $\{c, d\}$, there are three options for the contents of the cache of OPT if it does not fault. These are $\{b, c\}, \{a, c\}, \{a, d\}$. ALG inserts c into its cache and evicts a. At this point it has b and c in its cache.

If request 4 contains a, then ALG replaces b by a in its cache. It now has a and c. The configuration of OPT is either $\{a, c\}$ or $\{a, d\}$ (always assuming that OPT does not fault). There can be only one further request on which OPT does not fault, and that is a request which contains d. In this case, request 6 starts a new phase. We got a phase of length at most five.

If request 4 does not contain a, it must contain d (or we find a third independent set). Only one option for OPT remains, which is $\{a, d\}$. Using two uploads, ALG moves from $\{b, c\}$ to $\{a, d\}$ and request 5 starts a new phase. We got a phase of length four but cost five.

In both cases, ALG makes at most five uploads per at least one uploads of OPT, so it is at most 5-competitive.

Generalization for Arbitrary Values of k, u: We first build an algorithm for $u = 2$ and $k > 2$, using induction on k. We denote the algorithm which works on a cache of size k by ALG(k).

For $k = 3$, denote the first request in a phase by $\{a, b\}$ as before. Assume first that OPT has a in its cache. Then it has two 'free' places in its cache. For these places we can run the algorithm for $k = 2$. ALG(3) loads a in its cache and calls a modified version of ALG(2). This modified version runs for only one phase. Moreover, it knows that a is in the cache and thus will not fault on a request that contains a. When this modified version of ALG(2) returns, we know that OPT has made a fault (or is going to make a fault in the next step), *or* OPT did not load a after all. Now, ALG(3) loads b and again calls ALG(2). This time when it returns, we know that OPT has made a fault at some point, and we can start a new phase.

To improve the competitive ratio slightly, we can modify ALG(2) further so that it indicates whether the next phase should start with the last request that it processed (in case that this request was for a third independent set) or with the next request after that (in case that this last request fixes the contents of the cache of ALG in the last possible way such that OPT does not have a fault yet). We then get a phase cost of at most $12 = 1 + 5 + 1 + 5$. These costs are the cost for loading a, the first call to ALG(2), the cost for loading b and the second call to ALG(2), respectively.

Generally, we find that $R(\text{ALG}(k)) = 2 + 2 \cdot R(\text{ALG}(k-1))$, where we can take as base case $R(\text{ALG}(2)) = 5$. We find

$$R(\text{ALG}(k)) = 7 \cdot 2^{k-2} - 2 \quad \text{for } k \geq 2 \ .$$

Two remarks about this result:

- It is easy to give an algorithm of competitive ratio 2 for the case $k = 1$. This algorithm also works in phases, trying to guess the choice of OPT, that can be one of two pages in each phase. However, using $k = 1$ as a base case gives a ratio of 6 (instead of 5) for $k = 2$.
- It is vital that each (modified) algorithm ALG(i) that is called by another algorithm ALG($i+1$) knows the contents of the entire cache, in particular that part of it that it does not control. Otherwise ALG(i) could make too many faults by loading pages that are already in the cache elsewhere.

We can use a similar construction for $u > 2$. For the base case, we now do consider $k = 1$. The algorithm for this case loads the first page from the first request, say a_1 from request $\{a_1, a_2, \ldots, a_u\}$. On each fault after this, it loads the lowest-numbered page which is also in the new request, if possible. Note that a request which does not contain a certain page a_i immediately implies that OPT cannot have loaded a_i on the first request (unless it has a fault after this). Thus, as soon as we run out of pages from the first request in this way, we know that OPT must make (or has made) a fault and a new phase starts. This gives a u-competitive algorithm $\text{ALG}_u(1)$.

For the induction (on k), we call modified versions of simpler algorithms as before. In each induction step, we need to handle one request immediately (u times) before calling the simpler algorithm. Thus we find

$$R(\text{ALG}_u(k)) = u + u \cdot R(\text{ALG}_u(k-1)) = \frac{u^{k+1} - u}{u - 1}$$

since $R(\text{ALG}_u(1)) = u$. This proves the first half of Theorem 3.2.

3.3 Lower Bounds

In this section, we describe three different lower bounds for online algorithms. All lower bounds that we show use $u + k$ different pages. The request sequence is generated such that the online algorithm has a fault for every request, i.e., each request contains exactly the u pages that are absent from the cache of the algorithm.

Our lower bounds differ in the way that we define offline algorithms for this sequence. We begin by considering perhaps the simplest offline algorithm, which we will denote by OFF_1. OFF_1 checks which subset of u pages is least requested (in an arbitrarily long sequence of requests) and has the complement of this subset fixed in its cache. Each time the subset in question is requested anyway, it generates a cost of 2 for OFF_1: a cost of 1 to move to a different configuration (it is enough to

replace one page) and again 1 to move back (on the next request). Since there are $(k+u)!/(k!u!)$ possible subsets, this gives us a lower bound of $(k+u)!/(2k!u!)$. This gives a lower bound of 3 for $u = k = 2$ (we improve this specific value later). Note also that if k is constant and u grows without bound, this lower bound becomes $\Omega(u^k)$. This proves the second statement of Theorem 3.2, and shows that our algorithm from the previous section is optimal (up to a constant factor) for constant k.

For small u however, this lower bound can be improved. We first consider the case $u = 2$ in more detail. In this case, we use $k+2$ pages to construct the request sequence. We use a different offline algorithm OFF_2, and define phases in such a way that OFF_2 has only one upload per phase. Given a configuration of OFF_2, that it has just before a phase starts, we now define a phase as a consecutive subsequence of requests that fixes the configuration to which OFF_2 moves in the beginning of this phase, paying 1 or 0 (in case it does not move). We make sure that OFF_2 does not need to change its configuration until the beginning of the next phase. It is possible to show that it takes at least $2k$ requests before the configuration of OFF_2 is fixed. At this point, immediately after the phase, OFF_2 can move to a new configuration, determined by the requests of the next phase.

This gives a lower bound of $2k$ on the competitiveness of any online algorithm (which fails on every request, by our design). In particular, for the case $u = k = 2$, we find a lower bound of 4, only 1 less than our upper bound.

For $u = 2$ and $k > 2$, it is possible to improve on this further. We use a third type of offline algorithm OFF_3. Instead of continuing a phase until only one option for OFF_3 is left, we maintain two options for OFF_3 throughout and only fix OFF_3 after all phases have been defined. We define the very first phase to have only $2k-1$ requests, so that after this, OFF_3 still has (at least) two choices. Each later phase has $3k-2$ requests. It is possible to show that we can maintain the invariant that OFF_3 has two choices.

The state to which OFF_3 should move at the beginning of each phase is determined scanning the sequence of phases starting from the end, choosing each time a configuration for OFF_3 out of the two possible ones, which is a function of the next configuration chosen for it. We find a lower bound of $3k-2$.

We summarize some of the results obtained in the current section for $u = 2$ in the following table.

k	1	2	3	4	5
Lower bound	2	4	7	10	13
Upper bound	2	5	12	26	54

4 Paging with Bypassing

When bypassing is allowed, an algorithm is not forced to have in its cache a page from a request that it is going to service. Bypassing means that a page is used without loading it into the cache. The cost charged for this option is the same as for loading a page. Clearly, if a page is going to be used several times, it makes sense to load it into the cache. However, if a page is used only once, it is sometimes better to bypass it, so that the current contents of the cache can remain there. It is well known that for the standard problem, allowing bypassing increases the best competitive ratio by 1, i.e. it is $k+1$ for a cache of size k. Note that the best algorithms for this problem are marking algorithms, such as LRU, and FIFO (which is not a marking algorithm), and that their competitive ratio increases by 1. Thus the best online algorithms do not actually make use of the bypassing option.

Consider first the first lower bound presented in section 3.3. We can construct the lower bound sequence in the same manner. This means that if an online algorithm bypasses a certain request, that request is repeated until it is no longer bypassed. Having the option of bypassing, OPT does not need to pay 2 each time the request is the exact complement set, but just 1, to bypass on it (or more specifically, to bypass one page of this request). This gives a lower bound of $(k+u)!/(k!u!)$.

Interestingly, despite the recursive construction that we use, here it is also possible to design an algorithm with a competitive ratio which is only 1 higher than for the case without bypassing for any k. Here, in order to complete a phase, we must make sure that OPT had either a fault or one bypass during this phase (the phases are not shifted as in the proof of the algorithm without bypassing).

For some fixed value of k, consider the outer phase of the recursion. There are three cases. If OPT bypasses the very first request, it has a fault in the current (outer) phase and we are done for this phase. If OPT loads page i from this request ($i = 1, \ldots, k-1$), then by construction we know that it has a fault during the ith recursive call from the outer phase, or on the request which immediately follows it, which is also a part of the same outer phase. Finally, if OPT loads the kth page from this request, it can happen that it does not have a fault during the kth recursive call but instead on the very next request.

Therefore, we now construct the outer phase as before, but add one final request (without loading some page between the last recursive call and this request). This ensures that OPT has a fault in every outer phase. The inner phases can remain unchanged (we apply the old algorithm). This implies that the competitive ratio for any value of k increases by only 1.

This gives the following results for $u = 2$.

k	1	2	3	4	5
Lower bound	3	6	10	15	21
Upper bound	3	6	13	27	55

Thus for the case of bypassing, our algorithm is optimal for $u = k = 2$.

5 Resource Augmentation

Already in [14] the classical paging problem was studied in terms of resource augmentation. That is an extension of the usual competitive analysis, which allows an online algorithm to use greater resources than the optimal offline algorithm it is compared to. In this section, let h be the size of the cache used by an optimal offline algorithm and let $k > h$ be the size of cache used by an online algorithm. For standard online paging, the competitive ratio becomes constant if $h = \alpha k$ for fixed values of $\alpha < 1$ [14]. More precisely, it was shown in [14] that the best competitive ratio for this case is $\frac{k}{k-h+1}$.

In this section we focus on the case $u = 2$. We define a very simple algorithm which works in phases as before. The algorithm is defined for even values of k. The first $k/2$ requests are inserted completely into the cache. Note that this means that these first requests are independent. The next request starts a new phase (the cache contents are unmarked and all pages may be deleted). Let $\alpha = k/h$. We prove that for $\alpha > 2$, the algorithm has constant competitive ratio.

Theorem 5.1 *The competitive ratio of the above algorithm is at most* $\frac{2\alpha}{\alpha-2}$ *for even values of* k. *The competitive ratio tends to* 2 *as* α *grows.*

Proof: The sequence for a phase, including the first request of the next phase but not the first request of the current phase, contains k distinct pages. This holds since inside the phase at least $k-2$ new pages are requested and kept in the cache, and two additional pages are the first request of the next phase. The two pages of the first request are not equal to any of the other pages, therefore we have a total of $k+2$ pages. Out of this amount, OPT can have at most h in its cache after the first request. One spot in the cache is taken by a page of the very first request. It needs to service $k/2$ additional requests upto and including the first request of the next phase. This means that it has at least $k/2-h$ uploads. However, the algorithm has k uploads in each phase. □

For odd k, we need to define the algorithm slightly more carefully. In this case the first $\frac{k-1}{2}$ requests are inserted completely into the cache. On the request number $\frac{k+1}{2}$ of the phase, which is denoted $\{a,b\}$, only one page a is inserted into the cache. On the next request, if it contains page b, the algorithm evicts a and inserts b, and starts a new phase on the next request. If b does not belong to the next request, this request already starts a new phase. Using a proof very similar to the one above, it can be shown that the competitive ratio of this algorithm is $2\alpha'/(\alpha'-2)$ where $\alpha' = (k+1)/h$.

Proposition 5.2 *No online algorithm has a competitive ratio below 2 for any $k > 1$, even if $h = 1$.*

Proof: The sequence consists of requests as follows. The first request has two pages $\{a,b\}$. If the algorithm inserts both of them into the cache, the next request consists of two other pages. Otherwise, if only one page a is inserted into the cache, the next request is for the other page b together with a new page. This process is repeated. In the first option, OPT inserts one of the pages into its cache and in the second option it inserts b. In both cases the algorithm inserts two pages. □

Next, we focus on the smallest not trivial case, $h = 2$, $k = 3$. We can show that this algorithm improves on our algorithm for $h = k = 2$. We also design a lower bound.

Proposition 5.3 *The competitive ratio of the above algorithm for the case $h = 2, k = 3$ is at most 4. Any algorithm for $h = 2, k = 3$ has competitive ratio at least $5/2$.*

Proof: The number of uploads of the algorithm in one phase is at most 4. We need to show that OPT uploads at least once, on the second or third requests in the phase, or on the first request of the next phase. If after the second request OPT has not had a fault yet, it should have had one page of each request in its cache. If the next request is not a fault for OPT, it means that OPT should have b in its cache. At this time, the cache of the algorithm is a super set of the cache of OPT, and thus the next request must be a fault for OPT.

The lower bound sequence is constructed as before, using *five* distinct pages and forming the next request from the complement of the cache contents of the algorithm.

However, we continue our analysis in a subtly different way in order to get a better lower bound. The sequence is partitioned into sub-sequences in a way that sub-sequences of even index have length three, and ones of odd index have length two. We keep an invariant that inside a sub-sequence of length two, OPT has at least two options, and there exist such two options that share a common page. Inside a sub-sequence of length three, OPT has at least one option. The options for OPT are designed in a way that when it needs to move to a different cache state, it only needs to replace a single page, thus we make sure that in all sub-sequences, OPT has one fault.

Note that the state associated with a phase is one that OPT needs to move to in the beginning of the phase.

The sequence starts with a sub-sequence of length two. We assume OPT and ALG have the same pages in the cache. Consider first the situation for a sub-sequence of length two. By the invariant, OPT has at least one option for the previous phase (it also has one option if this is the very first phase), denote this option for the contents of the cache of OPT by $\{a_1, a_2\}$. Among the two requests in the subsequence, if one request contains a_1 or a_2, OPT can replace the other page by one of the pages of the other request. Otherwise, since there are only three other pages used in the sequence, there is some other page x common in the two requests, and OPT can replace either a_1 or a_2 with x. In both cases we get two distinct options for OPT that have one common page, as required. It can be that one of the options we allow for OPT is simply $\{a_1, a_2\}$.

Now, consider the situation for a sub-sequence of length three. OPT has at least two options for the previous phase by the invariant. Moreover, there exist such two options that have one page in common. Consider such two options. There are exactly three pages that participate in these two options and can be in the current cache of OPT, denote these three pages by a_1, a_2, a_3. Since the three requests in the next sub-sequence consist of six pages, there must be at least one page which is common in two requests (since the total number of distinct pages used for the sequence is five). Denote one of these pages by y.

Consider the case $y \neq a_i$ for $i = 1, 2, 3$. If y belongs to all three requests, this means that any state out of $\{\{y, a_i\} | i = 1, 2, 3\}$ is valid as a next state for OPT, and the invariant holds. If y belongs to only two requests, consider the third request which does not contain y. There is only one possible page except a_1, a_2, a_3, y, and thus this third request must contain one of a_1, a_2, a_3 (since it does not contain y and it contains two distinct pages). Let ℓ be the index of a page a_ℓ ($1 \leq \ell \leq 3$) that appears in the third request. Therefore, a valid next state for OPT is simply y, a_ℓ.

Finally, if the page in common in two requests is a_s for some $1 \leq s \leq 3$, and denote the third request by b_1, b_2, then OPT can move to either a_s, b_1 or a_s, b_2.

We now need to show that it is indeed always possible for OPT to choose a state it can move to in a beginning of a phase. Thus, OPT changes its state in the beginning of the sequence, and then after every phase.

We first compute the possible states for every phase. This computation is made from the beginning of the sequence forward towards the end. Using the construction above, we make a list of possible states of the cache of OPT for each phase.

Next, we compute the actual states to which OPT needs to move (i.e., we choose one state from the list per each phase). The computation is made backwards this time, starting from the end of the sequence. If there is more than one possible state for the very last phase, we choose one such state arbitrarily. OPT should move to this chosen state at the beginning of the last phase. Note that each state s in a list of a phase has an allowed state s' for the previous phase such that s' leads to s, i.e. an algorithm can move from s' to s using at most one fault. Therefore, we can each time choose such a valid state from the list arbitrarily, until we reach the beginning of the sequence. As defined above, the states we allow for the very first phase are both reachable from the initial phase using at most one fault.

We therefore proved the proposition. □

References

[1] Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1-2):203–218, 2000.

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

[3] Allan Borodin, Nathan Linial, and Michael Saks, An optimal online algorithm for metrical task systems, *Journal of the ACM*, 39:745–763, 1992.

[4] Marek Chrobak and Lawrence L. Larmore, The server problem and on-line games, *DIMACS Series in Discrete Math. and Theoretical Comp. Science*, vol. 7, 1992.

[5] Marc Demange and Vangelis Th. Paschos. On-line vertex-covering. *Theoretical Computer Science*, 332:83–108, 2005.

[6] Irit Dinur, Venkatesan Guruswami, Subhash Khot, and Oded Regev. A new multilayered PCP and the hardness of hypergraph vertex cover. *SIAM Journal on Computing*, 34(5): 1129–1146, 2005.

[7] Amos Fiat, Richard Karp, Michael Luby, Lyle A. McGeoch, Daniel Sleator, Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.

[8] Amos Fiat and Moty Ricklin. Competitive algorithms for the weighted server problem. *Theoretical Computer Science*, 130:85–99, 1994.

[9] Sandy Irani. Competitive analysis of paging. In A. Fiat and G. Woeginger, editors, *Online Algorithms: The State of Art*, pages 52–73. Springer, 1998.

[10] Anna Karlin, Mark Manasse, Lyle Rudolph, and Daniel Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[11] Mark Manasse, Lyle A. McGeoch, and Daniel Sleator, Competitive algorithms for server problems, *Journal of Algorithms*, vol. 11, pages 208-230, 1990.

[12] Lyle McGeoch and Daniel Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, 1991.

[13] Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and sub-constant error-probability PCP characterization of NP. In *Proc. 29th ACM Symp. on Theory of Comp.*, pages 475–484, 1997.

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