A Strongly Competitive Randomized Paging Algorithm

Lyle A. McGeoch *Daniel D. Sleator* *

23 March 1989

CMU-CS-89-122

* Partial support provided by DARPA (DOD), ARPA Order No. 4976, Amendment 20, monitored by the Air Force Avionics Laboratory under contract F33615-87-C-1499, and by the National Science Foundation under grant CCR-8658139. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF or the U.S. government.

A Strongly Competitive Randomized Paging Algorithm

Lyle A. McGeoch
Department of Mathematics
Amherst College
Amherst, MA 01002

Daniel D. Sleator *
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

March 23, 1989

Abstract

The *paging problem* is that of deciding which pages to keep in a memory of k pages in order to minimize the number of page faults. We develop the *partitioning algorithm*, a randomized on-line algorithm for the paging problem. We prove that its expected cost on any sequence of requests is within a factor of H_k of optimum. (H_k is the k^{th} harmonic number, which is about $\ln(k)$.) No on-line algorithm can perform better by this measure. Our result improves by a factor of two the best previous algorithm.

1. Introduction

The paging problem arises when trying to control a two-level memory system. Such a system has k pages of fast memory and $n - k$ pages of slow memory. A sequence of requests to pages is to be satisfied in their order of occurrence. In order to satisfy a request to a page, that page must be in fast memory. When a requested page is not in fast memory, a *page fault* occurs, and a page must be moved from fast memory to slow memory to make room for the new page to be put into fast memory. The *paging problem* is that of deciding which page to eject from fast memory. The cost to be minimized is the number of page faults.

Belady [1] gave a simple optimum algorithm for the paging problem. Belady's algorithm ejects the page that will remain unused for the longest time. This algorithm is *off-line*, using knowledge of future requests. An important class of paging algorithms are the *on-line* algorithms. These algorithms are not allowed use information about the future to process the pending request.

*Partial support provided by DARPA, ARPA order 4976, Amendment 20, monitored by the Air Force Avionics Laboratory under contract F33615-87-C-1499, and by the National Science Foundation under grant CCR-8658139.

Sleator and Tarjan [5] introduced the idea of comparing the performance of on-line paging algorithms with that of the off-line optimum. They showed that any on-line algorithm for the paging problem will, on some sequence of requests, have cost k times that of the optimum off-line algorithm.

A *randomized* paging algorithm is allowed to make use of a source of randomness when deciding what to do. Fiat *et al.* [2] extended the work of Sleator and Tarjan to the domain of randomized algorithms. To describe the results of that paper, it is useful to introduce the terminology of randomized competitiveness from Manasse *et al.* [3]. A randomized on-line algorithm is said to be *c-competitive* if on every sequence of requests its expected cost is within a factor of c (plus a constant) of that of every other algorithm, including those that are off-line. More formally, we let $C_A(\sigma)$ be the expected cost incurred by an algorithm A in processing a request sequence σ. An algorithm B is c-competitive if there exists a constant a such that for all algorithms A and all sequences σ,

$$C_B(\sigma) \leq c \cdot C_A(\sigma) + a.$$

The constant c is known as the competitive factor. An algorithm is said to be *strongly competitive* if it has the smallest possible competitive factor.

The marking algorithm of Fiat *et al.* [2] is a randomized algorithm for the paging problem. This algorithm is H_k-competitive if $k = n - 1$ and $2H_k$-competitive in general. (Here H_k denotes the k^{th} harmonic number: $H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$.) It is also shown that H_k is the smallest possible competitive factor for the paging problem. These results leave a gap of a factor of two between the lower bound on the competitive factor and what is achieved by an algorithm. The main result of this paper is a new algorithm whose competitive factor matches the lower bound, and is thus strongly competitive.

The paper has two main parts. The first part (Section 2) describes a way of maintaining a dynamically changing partition of the set of pages. We show that the partitioning procedure can be used to obtain a lower bound on the cost of any algorithm handling a given sequence of requests. The second part (Section 3) describes a randomized algorithm based on partitioning and analyzes its performance.

The paging problem is a special case of the *k-server problem*, the deterministic version of which was studied in Manasse *et al.* [3,4]. In this problem there is a set of n vertices numbered $1, 2, \ldots, n$, and a distance measure among them satisfying the triangle inequality. A collection of k mobile servers reside on these vertices. Given a sequence of requests, each of which specifies a vertex that requires service, the k-server problem is to decide how to move the servers in response to each request. If a requested vertex is unoccupied, then some server must be moved there. The requests must be satisfied in order of their occurrence in the request sequence. The cost of handling a sequence of requests is equal to the total distance moved by the servers.

In the *uniform k-server problem*, the cost of moving a server from any vertex to any other is one. The paging problem is isomorphic to the uniform k-server problem. The cor-

respondence between the two problems is as follows: the pages of address space correspond to the n vertices, and the pages in fast memory correspond to those vertices occupied by servers. In the remainder of this paper we shall use the terminology of the uniform k-server problem rather than that of the paging problem.

2. A Lower Bound on Optimal Cost

In this section we describe a dynamically changing labeled partition of the vertices. The partition evolves deterministically as a function of the request sequence, and can be maintained by an on-line algorithm. We show how to use this partition to obtain a lower bound on the cost incurred by any algorithm in satisfying the request sequence.

The partition is actually an ordered sequence of disjoint sets of vertices $S_\alpha, S_{\alpha+1}, \ldots S_{\beta-1}, S_\beta$ (some of which could be empty) whose union is the set $\{1, 2, \ldots n\}$. Each set S_i (except S_β) is labeled with an integer k_i. Initially $\alpha = 1$ and $\beta = 2$, S_1 is the set of vertices that are not initially covered by a server, S_2 is the set of vertices that are, and $k_1 = 0$. The numbers α and β increase over time.

The labels are related to the cardinalities of the sets and satisfy the following set of conditions, which we call the *labeling invariant*:

$$\begin{aligned} k_\alpha &= 0 & \\ k_i &> 0 & \alpha < i < \beta \\ k_i &= k_{i-1} + |S_i| - 1 & \alpha < i < \beta \\ k_{\beta-1} &= k - |S_\beta| & \end{aligned}$$

We will show that these conditions hold initially and that they are maintained as the partition evolves.

We can now describe how the labeled partition is updated in response to a request at a vertex v. Let i be such that $v \in S_i$. There are three cases:

Rule 1, $i = \beta$: Do nothing.

Rule 2, $\alpha < i < \beta$: First apply the following assignments:

$$\begin{aligned} S_i &\leftarrow S_i - \{v\} \\ S_\beta &\leftarrow S_\beta \cup \{v\} \\ k_j &\leftarrow k_j - 1 \qquad i \le j < \beta \end{aligned}$$

It might now be the case that some label is changed from one to zero. If this happens, let j be the largest integer such that $k_j = 0$. Now apply the following two assignments:

$$\begin{aligned} S_j &\leftarrow S_\alpha \cup S_{\alpha+1} \cup \cdots \cup S_{j-1} \cup S_j \\ \alpha &\leftarrow j \end{aligned}$$

Rule 3, $i = \alpha$: Do the following assignments:

$$\begin{aligned} S_\alpha &\leftarrow S_\alpha - \{v\} \\ S_{\beta+1} &\leftarrow \{v\} \\ k_\beta &\leftarrow k - 1 \\ \beta &\leftarrow \beta + 1 \end{aligned}$$

An easy induction shows that the labeling invariant remains satisfied after a request is processed in this way.

The following table shows the labeled partitions generated for a particular sequence of requests when $n = 9$ and $k = 6$. Each line shows the partition resulting after a request to the underlined vertex in the previous partition, as well as the rule that was applied to obtain the new partition. The leftmost set on a line is the current S_α and the rightmost set is the current S_β. Each set S_i (except S_β) is labeled with the appropriate k_i.

Initial Partition:	$\{7,8,\underline{9}\}_0 \quad \{1,2,3,4,5,6\}$
rule 3:	$\{7,8\}_0 \quad \{1,2,3,4,5,\underline{6}\}_5 \quad \{9\}$
rule 2:	$\{7,\underline{8}\}_0 \quad \{1,2,3,4,5\}_4 \quad \{9,6\}$
rule 3:	$\{7\}_0 \quad \{\underline{1},2,3,4,5\}_4 \quad \{9,6\}_5 \quad \{8\}$
rule 2:	$\{7\}_0 \quad \{2,3,4,5\}_3 \quad \{\underline{9},6\}_4 \quad \{8,1\}$
rule 2:	$\{7\}_0 \quad \{2,3,4,5\}_3 \quad \{\underline{6}\}_3 \quad \{8,1,9\}$
rule 2:	$\{7\}_0 \quad \{2,\underline{3},4,5\}_3 \quad \{\}_2 \quad \{8,1,9,6\}$
rule 2:	$\{7\}_0 \quad \{2,4,\underline{5}\}_2 \quad \{\}_1 \quad \{8,1,9,6,3\}$
rule 2:	$\{7,2,4\}_0 \quad \{8,1,9,6,3,5\}$

The partitioning procedure is significant because it permits an on-line algorithm to track the performance of an optimal off-line algorithm. The optimal off-line algorithm,

which we shall call OPT, was discovered by Belady [1]. This algorithm can be described as follows:

OPT: For each request to an uncovered vertex, move a server from the covered vertex for which the next request is farthest in the future. If two or more covered vertices are never requested again, move from the one with the higher number.

For completeness we now prove that OPT is optimal for any request sequence σ. Suppose A is an algorithm that starts in the same state as OPT, and makes the same moves as OPT for the first i requests of σ. We now show how A can be modified, without increasing its cost, so that it also makes the same moves as OPT on the $(i+1)^{st}$ request.

Suppose the request is at v, that OPT moves the server on w to v, and A moves a server on x to v. Define algorithm A' as follows: On the $(i+1)^{st}$ request, A' moves the server on w to v. A' now mimics the moves of A exactly, until one of two things happens: (1) A moves its server on w to another vertex u. Then A' moves the server on x to u, and A and A' are in the same state and have incurred the same cost. (2) There is a request at x, and A moves from u to x. Then A' moves from u to w. Algorithms A and A' are again in the same state, and the cost incurred by A' is at most that incurred by A. By the definition of OPT, we know that there must be a request to x before any request to w. This guarantees that A' is well defined and costs no more than A. By repeatedly modifying algorithm A in this manner, it can be transformed, without increasing its cost, into OPT. It follows that no algorithm A can handle σ more cheaply than OPT.

To streamline further discussion, we introduce some notation. Let

$$S_i^* = S_\alpha \cup S_{\alpha+1} \cup \cdots \cup S_{i-1} \cup S_i.$$

After processing $\sigma(1), \sigma(2), \ldots, \sigma(t)$ the algorithm OPT covers a particular set of vertices. This set can be computed using the partition and the remainder of the request sequence $\sigma(t+1), \sigma(t+2), \ldots$, as follows:

1. $S \leftarrow S_\beta$.
2. Sort the vertices in $S_{\beta-1}^*$ in order of earliest occurrence in $\sigma(t+1)\cdots$. That is, if a vertex v occurs in $\sigma(t+1)\cdots$ before another vertex w, then v comes before w in the sorted list. All vertices requested in $\sigma(t+1)\cdots$ come before all those that are not, and two vertices not requested are ordered by vertex number, lowest first.
3. Repeat the following step for each vertex v in the order defined in step 2, until S contains k vertices: Add v to S unless it would force S to contain more than k_i vertices from any set S_i^*.

Lemma 1 *At any point in the processing of a request sequence σ, the set S of vertices obtained by this procedure is the same as the set of vertices covered by OPT.*

Proof. The proof is by induction on the request sequence. The lemma clearly holds before the first request. Suppose the lemma is true after the first t requests of σ. We claim it holds after $t+1$ requests. If the next request is for a vertex in S_β, then OPT does not make a move, and the partition does not change. Since the new request was in S_β, the ordering of the vertices in $S^*_{\beta-1}$ does not change, and so S does not change.

If the next request is for a vertex v in S_α, then by the induction hypothesis, OPT is not covering v, and it moves the vertex requested farthest in the future. Rule 2 is applied to update the partition, *i.e.* β is incremented and v becomes the sole vertex in the new S_β. This also causes v to be added to S. Because S can only contain $k-1$ vertices from the new $S^*_{\beta-1}$, the new S is the same as the old S except that the vertex farthest in the future is dropped.

Now suppose that the next request is for a vertex v in S_i $(\alpha < i < \beta)$. It must be the case that $|S_\beta| < k$, so the set S contains at least one vertex from $S^*_{\beta-1}$. Vertex v is first in the sorted list of vertices from S^*_i, so it must be in S. By the induction hypothesis, OPT is already covering S and does not move. We must show that S does not change.

Before the partition is updated, S contains at most k_j vertices from each S^*_j. This means that S contains v and at most $k_j - 1$ other vertices from S^*_j when $i \le j \le \beta - 1$. When Rule 3 is applied to update the partition, v is moved from $S^*_{\beta-1}$ to S_β, and k_j is decremented for all j from i to $\beta - 1$. The decremented bounds offset the fact that v is no longer in S^*_j for any $j < \beta$. The net effect is that S does not change. □

Using this lemma, we can obtain a lower bound on the cost of any paging algorithm. Let $D(\sigma)$ denote the number of times a vertex in set S_α is requested during the processing of request sequence σ.

Theorem 1 *For any algorithm A and request sequence σ,*

$$C_A(\sigma) \ge D(\sigma).$$

Proof. Because OPT is optimal, the cost of algorithm A is no less than the cost of OPT. Lemma 1 characterized precisely the set of vertices covered by OPT. This set never includes any vertices in S_α, so OPT incurs a cost of one whenever such a vertex is requested. (The cost of OPT actually equals $D(\sigma)$, because OPT has no other costs.) □

3. An On-Line Algorithm

We can now describe the *partitioning algorithm*, a randomized H_k-competitive algorithm for the uniform k-server problem. It works by maintaining the labeled vertex partition described above, augmented with a system of *marks*. There will be one kind of mark (called an i-mark) for each set S^*_i, $\alpha < i < \beta$. There are k_i i-marks, which occupy distinct

vertices of S_i^*. These marks are only allowed on vertices in S_i or on vertices having an $(i-1)$-mark. The algorithm keeps a server on each vertex of S_β and on each vertex with a $(\beta-1)$-mark.

Given a labeled partition, call a vertex v *i-eligible* if it is in S_i or if it has an $(i-1)$-mark. Only i-eligible vertices may have an i-mark. By the labeling invariant there are exactly k_i+1 i-eligible vertices. We shall show later that any valid arrangement of marks is equally likely to be chosen by the partitioning algorithm.

We can now describe how the partitioning algorithm updates the marking. Initially $\alpha = 1$ and $\beta = 2$ and there are no marks of any kind.

When a request arrives for a vertex in S_β nothing happens to the partition, marks, or servers.

When a request arrives for a vertex v in S_i, $\alpha < i < \beta$, before applying Rule 2, we move marks around so that there is a j-mark on v for all j satisfying $i \le j < \beta$. This mark movement is done by repeating the following step for each j starting at i and ending at $\beta - 1$:

> If v has a j-mark then do nothing. Otherwise randomly choose some vertex w that has a j-mark. Transfer each l-mark (where $l \ge j$) from w to v.

When Rule 2 is applied, v ends up in S_β, and all the marks on v are erased. If α changes, all i-marks ($i \le \alpha$) are deleted. There are now the right number of marks of each type, confined to the right places. Recall that each $(\beta-1)$-mark corresponds to a server. If a $(\beta-1)$-mark is moved to v from some vertex, a server is also moved from the same vertex.

When a request arrives for a vertex v in S_α, we apply update Rule 3. We then create $k-1$ new $(\beta-1)$-marks and distribute them randomly among the k $(\beta-1)$-eligible vertices. These eligible vertices are exactly those covered by a server before the request. The one of these that is chosen not to have a $(\beta-1)$-mark is the vertex from which the server is moved.

The number of different valid arrangements of marks is $\prod_{\alpha \le i < \beta}(k_i+1)$, because there are exactly (k_i+1) ways to place the i-marks. The following lemma shows that while running the partitioning algorithm, each valid arrangement of marks is equally likely.

Lemma 2 *The partitioning algorithm is equally likely to produce each valid arrangement of marks.*

Proof. We shall prove the lemma inductively. Clearly it is true initially, when there are no marks or eligible vertices. Now suppose the assertion is true before a request to a vertex v. The remainder of the proof shows that the lemma remains true after the request to v is processed.

If $v \in S_\beta$, nothing happens to the partition or the marks, so the lemma remains true.

If $v \in S_\alpha$, Rule 3 is applied and β is incremented. No i-marks are moved for any $i < \beta - 1$, so the distribution of these marks is unchanged. The algorithm introduces $k_{\beta-1}$ new $(\beta-1)$-marks, randomly placed on the $k_{\beta-1}+1$ eligible vertices.

This leaves the case where $v \in S_i$ and $\alpha < i < \beta$. In this case the j-marks, where $\alpha < j < i$, are not changed by the partitioning algorithm, so their distribution remains the same. For the j-marks with $j \geq i$, the situation is more complex. The action of this step of the partitioning algorithm can be broken into two parts. The first part loads vertex v with j-marks for all $j \geq i$. The second part moves v into S_β and removes all of the marks on v. We claim that after the first part of the process, the arrangement of marks is equally likely to be any valid arrangement that obeys the additional constraint that vertex v has a j-mark for all $j \geq i$. It immediately follows that the induction hypothesis is maintained after the second part of the step.

It remains to prove our claim. Call an arrangement of marks (i, j, v)-constrained if it is a valid arrangement of marks and if vertex v has marks of types $\{l \mid i \leq l \leq j\}$. (It may have other marks as well.) The initial arrangement of marks is equally likely to be any arrangement that is $(i, i-1, v)$-constrained. It is easy to verify that the j^{th} step of the mark-moving process transforms a random $(i, j-1, v)$-constrained arrangement into a random (i, j, v)-constrained arrangement. An induction on j proves our claim. □

We use Lemma 2 to prove the following lemma, which bounds the probability that a server moves on a request that invokes Rule 2.

Lemma 3 *While processing a request for vertex $v \in S_i$, where $\alpha < i < \beta$, the probability that the partitioning algorithm moves a server is at most*

$$\sum_{i \leq j < \beta} \frac{1}{k_j + 1}.$$

Proof. Recall that the mark-moving procedure works by iterating over levels from i up to $\beta - 1$. For each level, the iteration step ensures that v receives an i-mark. Eventually v receives a $(\beta-1)$-mark, which corresponds to a server. The probability that a server moves is bounded above by the expected number of iterations on which marks move. The probability that a move occurs on step j equals the probability that v still has no j-mark after the $(j-1)^{st}$ step. As shown in Lemma 2, the j^{th} step begins in a random $(i, j-1, v)$-constrained arrangement. In this arrangement, the probability that v has no j-mark is $1/(k_j+1)$. This is the probability that a move occurs on the j^{th} step. Summing of over all j, we obtain the desired bound. □

Theorem 2 *The partitioning algorithm is an H_k-competitive algorithm for the uniform k-server problem.*

Proof. We shall prove this theorem using the following potential function:

$$\Phi = \sum_{\alpha \le i < \beta} (H_{k_i+1} - 1).$$

This function is initially zero and is always non-negative. We shall prove by induction (on the length of σ) the following *potential invariant*:

$$C_{PA}(\sigma) + \Phi \le H_k \cdot D(\sigma).$$

Here $C_{PA}(\sigma)$ is the expected cost incurred by the partitioning algorithm on a sequence σ, and $D(\sigma)$ (as in Theorem 1) is the number of occurrences in σ of requests to vertices in S_α. Theorem 1 in conjunction with the potential invariant proves the theorem. It remains to prove this invariant by induction.

The invariant is clearly true initially, because all quantities are zero. We shall show that if it holds before a request to v, then it holds after it. As usual, there are three cases to consider.

If $v \in S_\beta$, then none of the three terms change, and the inequality remains true.

If $v \in S_i$, $\alpha < i < \beta$, then the right side of the potential invariant does not change. We shall show that the left side does not increase. The expected cost of this request to the partitioning algorithm is just the expected number of servers moved. A bound on this quantity is given in Lemma 3:

$$\text{expected cost of the request to PA} \le \sum_{i \le j < \beta} \frac{1}{k_j + 1}.$$

The potential function also changes as a result of an application of Rule 2. We can write this change as a function of the values of the labels before and after Rule 2 is applied. The primed symbols represent the values after the update rule, and the unprimed represent those before the update.

$$\begin{aligned}
\Delta\Phi &= \sum_{\alpha' \le j < \beta} \left(H_{k'_j+1} - 1\right) - \sum_{\alpha \le j < \beta} \left(H_{k_j+1} - 1\right) \\
&\le \sum_{\alpha \le j < \beta} \left(H_{k'_j+1} - 1\right) - \sum_{\alpha \le j < \beta} \left(H_{k_j+1} - 1\right) \\
&= \sum_{i \le j < \beta} \left[(H_{k_j} - 1) - (H_{k_j+1} - 1)\right] \\
&= \sum_{i \le j < \beta} \left[(H_{k_j} - 1) - (H_{k_j} + \frac{1}{k_j + 1} - 1)\right] \\
&= -\sum_{i \le j < \beta} \frac{1}{k_j + 1}
\end{aligned}$$

This cancels the expected cost of the request, and finishes the case.

The final case to consider is when $v \in S_\alpha$. In this case the right side of the potential invariant increases by H_k, and the expected cost incurred by the partitioning algorithm is just one. To verify the invariant we have to show that the potential increases by at most $H_k - 1$. The change in the potential is in fact exactly $H_k - 1$. This is because Rule 3 does not change any k_j's, but it adds a new k_β, whose value is k - 1. The added term in the potential is $H_{k_\beta+1} - 1 = H_k - 1$. □

References

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

[2] Fiat, A., Karp, R. M., Luby, M., McGeoch, L. A., Sleator, D. D., and Young, N. E. Competitive paging algorithms. Carnegie Mellon University Computer Science technical report CMU–CS–88–196, 1988. Submitted for publication.

[3] Manasse, M. S., McGeoch, L. A. and Sleator, D. D. Competitive algorithms for on-line problems. In *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 322–333, Chicago, 1988.

[4] Manasse, M. S., McGeoch, L. A. and Sleator, D. D. Competitive algorithms for server problems. Carnegie Mellon University Computer Science technical report CMU–CS–88–197, 1988. *J. Algorithms*, to appear.

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