Yale University
Department of Computer Science

Randomized Algorithms for the List Update Problem

Nick Reingold[1] *Jeffery Westbrook*[2]

YALEU/DCS/TR-804
June 1990

[1]Department of Computer Science, Yale University, New Haven, CT 06520-2158. Research partially supported by NSF grants CCR-8808949 and CCR-8958528.
[2]Department of Computer Science, Yale University, New Haven, CT 06520-2158.

Randomized Algorithms for the List Update Problem

*Nick Reingold** *Jeffery Westbrook*†

Abstract

We prove upper and lower bounds on the competitiveness of randomized algorithms for the list update problem of Sleator and Tarjan. We give a simple and elegant randomized algorithm, BIT, that is more competitive than the best possible deterministic algorithm. Among randomized algorithms for request-answer games, BIT is the first that uses only a bounded number of random bits, independent of the number of requests. We also prove lower bounds on list-update algorithms against oblivious and adaptive on-line adversaries.

1 Introduction

Recently much attention has been given to *competitive* analysis of on-line algorithms [3, 7, 8, 11]. In their seminal work on competitive analysis [11], Sleator and Tarjan studied the move-to-front algorithm, a heuristic commonly used in system software to maintain a set of items as an unsorted linear list. This problem is called the *list update* problem. The cost of accessing an item is determined by its distance from the front of the list, and the list may be rearranged during the processing of a sequence of requests. Sleator and Tarjan demonstrated that the move-to-front algorithm is 2-competitive. Roughly speaking, an on-line algorithm is c-competitive if, for any request sequence, its cost is no more than c times the cost of the optimum off-line algorithm for that sequence. Subsequently Karp and Raghavan [private communication, 1990] noted that no deterministic algorithm for the list update problem can be better than 2-competitive, so in a very strong sense move-to-front is as good as any deterministic on-line algorithm.

A great deal of recent work has focused on the use of randomization to improve the competitiveness of on-line algorithms. Randomization has been used very effectively in the page caching problem; there are several randomized algorithms that are

*Department of Computer Science, Yale University, New Haven, CT 06520-2158. Research partially supported by NSF grant CCR-8958528.

†Department of Computer Science, Yale University, New Haven, CT 06520-2158.

exponentially more competitive than the best possible deterministic algorithms [5]. In this paper we examine the effect of randomization on algorithms for list update.

In Section 2 we define the problem more precisely and discuss the c-competitive performance measure for on-line algorithms. We also discuss the issue of *adversaries* for randomized algorithms. In Section 3 we present a very simple randomized algorithm, BIT, that is 1.75-competitive against oblivious adversaries. The best previous result was a 1.875-competitive algorithm due to Irani [private communication, 1990]. Our BIT algorithm is interesting not only for its simplicity and speed, but also because it makes random choices only during an initialization phase, using exactly n random bits, where n is the number of items in the list. After this initialization phase, BIT runs deterministically. The basic BIT algorithm can be improved to yield a slightly better competitive factor, while still using $O(n)$ random bits. In Sections 4 and 5 we turn to the question of lower bounds for randomized algorithms. In Section 4 we show that no randomized algorithm can be better than 2-competitive against an adaptive on-line adversary. Thus list update is a new example of a case where the adaptive off-line and on-line adversaries are equally powerful. In Section 5 we give some lower bounds for randomized algorithms against an oblivious adversary. The best previous bound was 1.18, due to Karp and Raghavan [private communication, 1990]. We extend their technique to show a lower bound of 1.27. This still leaves a substantial gap between the upper and lower bounds. We close with some comments and open questions in Section 6.

2 List Update and Competitive Algorithms

The list update problem is that of storing a dictionary as a linear list. A *request* is either an access to an item, an insertion of an item, or a deletion of an item. The algorithm that maintains the list may rearrange it at any time via an *exchange* of two adjacent items. A list update algorithm must search for an accessed item by starting at the front of the list and inspecting each item in turn until the requested item is found. An insertion is done by searching the entire list to ensure that the item is not already present, and then inserting the new item at the back of the list. A deletion is done by searching for the item and then removing it. At any time, the algorithm may exchange the position of any two adjacent items in the list. Each request or exchange has a cost.

More formally, in the list update model as defined by Sleator and Tarjan, an algorithm *services* a request by doing the following three things:

1. The algorithm makes any sequence of exchanges. Each exchange costs 1. These exchanges are called paid exchanges.

2. The algorithm receives and immediately services the next request. An access or deletion of the ith item in the list costs i. An insertion costs $n+1$, where n is the length of the list prior to the insertion.

3. The algorithm moves the accessed or inserted item any distance forward in the list. These exchanges cost nothing and are called free exchanges.

The cost of servicing the request is the sum of the costs of the paid exchanges and cost of the access, insertion, or deletion that is done. For a list update algorithm, A, we define the *cost* of servicing a sequence of requests σ to be the sum of all costs incurred in servicing each request, in turn, in the sequence. We write $\text{Cost}(A, \sigma)$ for this cost. Since insertions and deletions can be regarded as special cases of accesses, we will be mainly interested in sequences that consist only of accesses to a fixed-size list. If there are n items in the list, we assume they are named by the numbers from 1 to n. Our results will be appropriately extended to the general case at the end of our discussion.

We say a list update algorithm is *on-line* if it must service each request without any knowledge of future requests. We say the algorithm is *off-line* if at all times it knows the entire sequence of requests. Following [3, 7, 8, 11] we define a performance measure for on-line algorithms known as *competitiveness.*

Definition A deterministic list update algorithm, A, is *c-competitive* if there is a constant k such that for all list sizes, n, all off-line algorithms, $\hat{A}$, and all request sequences $\sigma \in \{1, 2, \ldots, n\}^*$,

$$\text{Cost}(A, \sigma) \leq c \cdot \text{Cost}(\hat{A}, \sigma) + k$$

Definition The *competitive ratio* of an algorithm, A, is the infimum of all c for which A is c-competitive.

Sometimes it is convenient to restrict our attention to lists of a fixed size. Therefore, we define the notions of *c-competitive on lists of size n*, and the *competitive ratio for lists of size n* in the obvious ways.

Several algorithms for list update have been studied [2, 10, 11]. The move-to-front algorithm uses the following simple heuristic: after an access or insertion of item x, move x to the front of the list using free exchanges. No other exchanges are done. Sleator and Tarjan showed that move-to-front is 2-competitive. In fact, under a more general cost measure in which the cost to access the i^{th} item in the list is $f(i)$ and the cost of a paid exchange of the i^{th} and $i + 1^{st}$ items is $f(i + 1) - f(i)$, move-to-front remains 2-competitive as long as $f(i)$ is convex. In Section 4 we will derive a lower bound of 2 on the competitiveness of any deterministic algorithm. It is natural to ask if a randomized algorithm can do better.

For randomized list update algorithms, the definition of competitiveness is somewhat complicated. The competitiveness of an algorithm is defined with respect to an *adversary.* Two factors differentiate adversaries: how the request sequences are generated, and how the adversary is charged for servicing the sequence. Following [8] and [1] we consider three kinds of adversaries:

Oblivious Adversary: The oblivious adversary generates a request sequence before the on-line algorithm begins to process it. The oblivious adversary will be charged the cost of the optimum off-line algorithm for that sequence.

Adaptive On-Line Adversary: This adversary is allowed to watch the on-line algorithm in action, and base the next request on all previous moves made by the on-line algorithm. The adversary must also service the requests on-line, however. The sequence of events is this (for each request): 1) the adversary generates a request, 2) the adversary services the request, and 3) the on-line algorithm services the request.

Adaptive Off-Line Adversary: This adversary generates the requests adaptively, as above, but is charged the minimal (off-line) cost for the sequence.

Definition A randomized on-line algorithm, A, is *c-competitive against Oblivious (respectively Adaptive On-Line, Adaptive Off-Line) adversaries* if there is a constant k such that for all list sizes, n, all Oblivious (respectively Adaptive On-Line, Adaptive Off-Line) adversaries, $\hat{A}$, and all request sequences $\sigma \in \{1, 2, \ldots, n\}^*$,

$$\mathrm{E}\left[\mathrm{Cost}(A, \sigma) - c \cdot \mathrm{Cost}(\hat{A}, \sigma)\right] \leq k$$

The above expectation is over the random choices made by the on-line algorithm.

Definition We define the *competitive ratio against Oblivious (respectively Adaptive On-Line, Adaptive Off-Line) adversaries* for a randomized on-line algorithm, A, as the infimum of all c for which A is c-competitive against Oblivious (respectively Adaptive On-Line, Adaptive Off-Line) adversaries. We also make the obvious definitions for competitiveness for a fixed size list.

The oblivious, adaptive on-line, and adaptive off-line adversaries are sometimes called the weak, medium, and strong adversaries, respectively. If a randomized algorithm is c-competitive against oblivious adversaries then for any sequence σ, the expected cost of processing σ using algorithm A is no more than c times the cost of the optimum off-line algorithm plus some constant k.

The first example of a randomized algorithm for list update with a competitive ratio provably less than two against an oblivious adversary was given by Irani [private communication, 1990], using techniques different from ours. That algorithm is 1.875-competitive.

The adaptive adversaries in some sense model a situation in which the random choices made by the algorithm have an effect on the future composition of the request sequence. It is perhaps worth noting that there is no known application of competitive analysis in which an adaptive adversary seems the most natural model.

3 The BIT Algorithm

In this section we describe and analyze a very simple randomized algorithm that we call "BIT", which is 1.75-competitive.

Essentially, BIT is "move-to-front every other access". Each item in the list has an associated one-bit counter. The item is in state 1 or 2 if the value of the counter is 0 or 1, respectively. Prior to processing any requests, a string of n bits is chosen uniformly at random from all possible n bit strings, and the counters of each item in the list are initialized according to the bits in the string, i.e., the counter of item i is initialized to the value of the i^{th} bit in the string. After this initialization, BIT services the request sequence deterministically as follows. Suppose item x is accessed. If item x is in state 2 it remains in the same position and enters state 1 by decrementing its counter mod 2; if x is in state 1 it moves to the front of the list and enters state 2 by decrementing its counter mod 2.

Note that the initial value of each counter is 0 or 1 with equal probability, and hence after j accesses to item x the value of its counter is still 0 or 1 with equal probability. We now analyze the competitiveness of this algorithm against an oblivious adversary.

Theorem 3.1 *Let σ be any sequence of m accesses to an n item list, and let OPT be any deterministic off-line algorithm that services σ. Assuming that BIT and OPT begin with the same initial list, the expected cost of the BIT algorithm on σ is at most* $1.75\text{Cost}(OPT, \sigma) - 3m/4$.

This theorem immediately implies that BIT is 1.75-competitive against an oblivious adversary, since such an adversary chooses a request sequence ahead of time, with no knowledge of BIT's random choices, and is charged the optimum cost among off-line algorithms for that sequence. (Several algorithms may achieve the optimum cost.)

Lemma 3.2 *At the time of the j^{th} access in σ, the probability that the state of item x is either 1 or 2 is independent of the position of x in OPT's list, and therefore is exactly* $1/2$ *for all x.*

Since OPT is deterministic, x always occurs at position i_j in OPT's list at the time of the j^{th} access. The state of item x is equiprobably 1 or 2. Thus the location of x in OPT's list conveys no information about the state of x. This argument can be made more rigorous by examining the sample space associated with the events "item x is in state s and at position i in OPT's list", for $s \in \{1, 2\}$ and $1 \leq i \leq n$. This lemma is not true if the adversary is adaptive. The positions of the items in OPT's list convey information about the access sequence, and in the case of an adaptive adversary the access sequence may convey information about the states of the items, since the adversary may choose accesses based on those states.

Our analysis extends the approach used by Sleator and Tarjan on the basic move-to-front heuristic. We bound the cost to BIT in terms of the cost to OPT using a potential function. An *inversion* is an ordered pair of items (y, x) such that x occurs before y in the list maintained by OPT (the optimum ordering of the pair) while x occurs after y in the list maintained by BIT. The set of inversions changes with time as BIT and OPT rearrange their lists. The cost to BIT of an access to item x is given by the cost of the access to OPT plus the number of inversions (w, x) for some w minus the number of inversions (x, z) for some z.

We define a potential function Φ that maps a two-tuple consisting of BIT's list and OPT's list to the integers. An inversion (y, x) is called a *type 1* inversion if x is in state 1 and is called a *type 2* inversion if x is in state 2. The type of the inversion (y, x) is the number of accesses to x before x next moves to front. Let ϕ_1 denote the number of type 1 inversions and ϕ_2 denote the number of type 2 inversions. We define $\Phi = 2\,\phi_2 + \phi_1$. The amortized cost of an operation performed by either OPT or BIT is defined to be the actual cost t of the operation to BIT plus the change in the potential function, $\Delta\Phi$. Since BIT is randomized, t and $\Delta\Phi$ are random variables. Note that Φ is initially zero, since both OPT and BIT begin with their lists in the same configuration, and that Φ is always non-negative. By a standard property of potential function analysis, this implies that if the amortized cost of the ith operation is bounded by a_i, the total cost to BIT is bounded by $\sum a_i$, and by linearity of expectations $\mathrm{E}[\mathrm{Cost}(\mathrm{BIT}, \sigma)] \leq \sum \mathrm{E}\,[a_i]$.

Consider the amortized cost of an access to x. The cost of this access to the optimum algorithm is k, where k is the position of x in OPT's list. Let R be a random variable that counts the number of inversions (w, x) for some w at the time of the access. The actual cost to BIT is at most $k + R$. We write $\Delta\Phi = A + B + C$, where A is a random variable giving the change in potential due to new inversions created during the access, B is a random variable giving the change in potential due to old inversions removed during the access, and C is a random variable giving the change in potential due to old inversions that change type during the access.

Suppose that $R = r$ and consider the value of $B + C$. If x is in state 2 then it stays in place; $B = 0$ and $C = -r$, since each inversion (w, x) goes from type 2 to type 1. If x is in state 1 then it moves to front and removes all inversions (w, x); $B = -r$ and $C = 0$ since each inversion is of type 1. In both cases $B + C = -r = -R$ and hence the expected amortized cost of the access is at most $k + \mathrm{E}[A]$.

New inversions can only be created when x is in state 1 and so moves to front. In the worst case a new inversion (x, z_i) is created for each of the $k-1$ items z_i preceding x in OPT's list. Each new inversion (x, z_i) increases the potential by 1 or 2 depending on its type, which in turn depends upon the state of z_i. We write

$$\mathrm{E}[A] \leq \mathrm{E}[\sum_{1 \leq i < k} Z_i] = \sum_{1 \leq i < k} \mathrm{E}[Z_i] \tag{1}$$

where Z_i is a random variable that measures the cost of a new inversion (x, z_i). To upper bound the expected value of Z_i we ignore the possibility that the inversion

(x, z_i) already exists, since this only decreases the expected value. Variable Z_i is 0 if x is in state 2, is 1 if x is in state 1 and z_i is in state 1, and is 2 if x is in state 1 and z_i is in state 2.

By Lemma 3.2 the probability that x is in either state is exactly $1/2$, as is the probability that z_i is in either state. Thus we can write

$$\mathrm{E}[Z_i] \leq \frac{1}{2}(0) + \frac{1}{2}(\frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 1) = \frac{3}{4} \tag{2}$$

This implies by equation (1) that the expected value of A is $3(k-1)/4$, and hence the expected cost of the access to BIT is no more than $1.75k - 3/4$.

The cost of a paid exchange to BIT is similarly bounded. OPT pays 1 for the exchange. In the worst case the exchange creates an inversion (y, x). Again applying Lemma 3.2, this inversion increases the value of Φ by 2 with probability $1/2$ (the probability that x in state 0) and by 1 with probability $1/2$. Hence the expected cost to BIT is 1.5 times the cost to OPT.

At this point we have proven Theorem 3.1, since by a theorem of Reingold and Westbrook [9], for any request sequence σ there is an optimum algorithm that does only paid exchanges. For thoroughness, however, we consider free exchanges performed by OPT. A free exchange can only occur immediately after an access to some item, say x, and must move x forward past some other item, say y. The actual cost of the exchange is zero to both OPT and BIT. If x was in state 0 at the time of the access, then x is at the front of BIT's list at the time of the free exchange. Therefore, the free exchange removes the inversion (x, y). By Lemma 3.2, the cost of this inversion is 1 with probability $1/2$ and 2 with probability $1/2$. If, however, x was in state 2 at the time of the access then at the time of the free exchange by OPT, x is in state 1, so the free exchange either removes an inversion, or create one of type 1. Therefore, the expected change in potential is no more than $(1/2)((1/2)(-1) + (1/2)(-2)) + (1/2)1 = -1/4$. This concludes the proof of Theorem 3.1. Notice that BIT uses n random bits regardless of the length of the request sequence. An interesting open question is whether this is possible for randomized algorithms in other on-line situations.

The extension of BIT to handle insertions and deletions is straightforward. On an insertion, BIT randomly initializes the counter of the new item, and then moves the new item to the front if it is in state 1. Suppose the list has size n. The insertion is clearly equivalent to the first access to the $(n+1)^{st}$ item in a list of size $n+1$. Thus the analysis applied above to accesses can be applied here to show that an insertion has the same expected cost as an access. (This observation is due to Sleator and Tarjan[11].) Similarly, a deletion is like an access, except that all inversions involving the deleted item disappear, so the potential function decreases even more than during an access.

Corollary 3.3 *Let σ be any sequence of m accesses, insertion, and deletions to an initially empty list, and let OPT be any deterministic off-line algorithm that services σ. The expected cost of the BIT algorithm on σ is at most* $1.75\mathrm{Cost}(OPT, \sigma) - 3m/4$.

It is possible to modify the basic BIT algorithm to improve, somewhat, the competitive ratio, at the expense of making the algorithm more complicated. Let s be a positive integer, and S be any nonempty subset of $\{0, 1, \ldots, s-1\}$. The algorithm COUNTER(s,S) keeps a mod s counter for each item. Initially, each counter is randomly set to some number in $\{0, 1, \ldots, s-1\}$, each value chosen independently and with equal probability. At a request to item x, the algorithm pays for the access, moves x to the front if and only if $x \in S$, and then decrements the counter mod s. BIT is COUNTER(2, $\{0\}$). Any COUNTER algorithm has the property that for a fixed size list, it uses a fixed number of random bits regardless of the size of the request sequence.

Theorem 3.4 *COUNTER(s, S) is* $\max\{\sum_{j=1}^{s-1} jp_j, 1 + p_1 \sum_{j=1}^{s-1} jp_j\}$*-competitive*

The analysis of COUNTER(s, S) is very similar to the analysis of BIT, so we will just sketch the differences. For $i = 0, 1, \ldots, s-1$, let $d(i)$ be the number of accesses before an item with counter value i next moves to the front. For an item x, let $c(x)$ be the number of accesses to x before x moves to the front. For $j = 1, 2, \ldots, s$, let $p_j = (1/s)|\{i : c(i) = j\}|$; this is the probability that an item will next move to front after j accesses. After the initialization phase and before any accesses, $\Pr[c(x) = j]$ is just p_j for any item x. The analogue of Lemma 3.2, namely that at any time, the probability that $c(x) = j$ is p_j, independent of the position of x in OPT's list, is easily verified.

An inversion (y, x) is of *type* j if $c(x) = j$. Let ϕ_j denote the number of inversions of type j. Our potential function is $\Phi = \sum_{j=1}^{s} j \cdot \phi_j$.

Consider the expected amortized cost of an access to x. If x does not move to the front then, since $c(x)$ decreases by one, the decrease in potential due to inversions which change type is exactly the number of inversion (w, x). If x moves to the front the number of inversions destroyed is exactly the number of inversions (w, x). In either case the expected amortized cost of the access is at most the position of x in OPT's list plus the expected cost of new inversions created. It is not hard to see that the expected cost of the inversions created is at most $(k-1)p_1 \sum_{j=1}^{s} jp_j$ (where k is the position of x in OPT's list), so the cost to COUNTER(s, S) is at most $1 + p_1 \sum_{j=1}^{s} jp_j$ times the cost to OPT.

For a paid exchange by OPT, the worst case is that one new inversion is created. The expected increase in potential is $\sum_{j=1}^{s} jp_j$. Therefore we have that in this case the cost to COUNTER(s, S) is no more than $\sum_{j=1}^{s} jp_j$ times the cost to OPT. This completes the analysis of COUNTER(s,S).

For some choices of s and S, we can get a better competitive factor than 1.75. For example, COUNTER(7,$\{1, 3, 5\}$) is 85/49-competitive ($\approx$ 1.735-competitive).

4 Lower Bounds Against Adaptive Adversaries

In [1], Ben-David, *et al.*, show that for a wide variety of *request-answer games*, of which the list update problem is a special case, an adaptive off-line adversary is so powerful that randomization is no help against it. That is, they show that if there is a randomized algorithm which is c-competitive against an adaptive off-line adversary, then there is a deterministic algorithm which is c-competitive. In this section we will prove an analogous result for on-line list update algorithms against adaptive on-line adversaries.

The key to the proof is the following lemma, which says that an on-line algorithm can significantly reduce the expected cost per access provided it has advance information about the frequency of items in the request sequence.

Lemma 4.1 *Let A be an on-line algorithm maintaining a list of size n. Suppose that A will be asked to service a sequence of accesses, $\langle r_1, r_2, \ldots, r_m \rangle$, drawn according to some probability distribution on all access sequences of length m. If A is given in advance e_i, the expected number of times item i will be accessed, for all items i, then A can arrange that*

$$\mathrm{E}\left[\mathrm{Cost}(A, \langle r_1, r_2, \ldots, r_m \rangle)\right] \leq \frac{n(n-1)}{2} + \frac{m(n+1)}{2}.$$

Notice that this implies that for every $\epsilon > 0$ and for sufficiently large m, A can arrange that its average cost per access is less than $(n+1)/2 + \epsilon$.

Proof Suppose that the jth possible request sequence has probability p_j of being chosen, and that $f_j(i)$ is the number of times that item i appears in the jth request sequence. Notice that

$$\begin{aligned} e_i &= \sum_{j=1}^{n^m} p_j f_j(i), \qquad \text{and} \\ \sum_{i=1}^{n} e_i &= m. \end{aligned}$$

Now, suppose that prior to servicing the access sequence A arranges its list in order of decreasing e_i. This costs it no more than $n(n-1)/2$. We may assume that after A does this the list looks like $1, 2, \ldots, n$, so that $e_1 \geq e_2 \geq \cdots \geq e_n$. If A never makes any more exchanges, its expected cost on any request sequence of length m is

$$\begin{aligned} \sum_{j=1}^{n^m} p_j \sum_{i=1}^{n} i f_j(i) &= \sum_{i=1}^{n} i \sum_{j=1}^{n^m} p_j f_j(i) \\ &= \sum_{i=1}^{n} i e_i \end{aligned}$$

$$\leq \left(\frac{1}{n}\right)\left(\sum_{i=1}^{n} i\right)\left(\sum_{i=1}^{n} e_i\right) \qquad (3)$$
$$\leq \frac{m(n+1)}{2}.$$

The inequality in (3) follows from Chebyshev's summation inequalities (see Theorem 43 of [6]). Therefore, A's expected total cost is no more than $n(n-1)/2+m(n+1)/2$. ∎

Corollary 4.2 *Let A be an on-line algorithm maintaining a list of size n. Suppose that A will be asked to service request sequence $\langle r_1, r_2, \ldots, r_m \rangle$. If A is given, in advance, e_i, the number of times item i will be requested, for all items i, then A can arrange that*

$$\text{Cost}(A, \langle r_1, r_2, \ldots, r_m \rangle) \leq \frac{n(n-1)}{2} + \frac{m(n+1)}{2}$$

Proof Immediate from Lemma 4.1. ∎

Using this corollary, we can now derive the lower bounds for deterministic on-line algorithms and for randomized on-line algorithms against adaptive off-line adversaries. The former is an unpublished result of Karp and Raghavan, and the latter is a special case of the result of Ben-David, *et al.* [1] mentioned earlier.

Theorem 4.3 *If A is deterministic and c-competitive then $c \geq 2$. If A is randomized and c-competitive against adaptive off-line adversaries then $c \geq 2$.*

Proof In either case, an adversary can always construct a long request sequence on which A pays n per access. Again in either case, the adversary knows the frequency count of each item in the request sequence *before* it must process the sequence. By Corollary 4.2 the adversary can arrange to pay arbitrarily close to $(n+1)/2$ per access, so A can't be better than $2n/(n+1)$-competitive. Since this must be true for all n, A can't be better than 2-competitive. ∎

Notice that if we fix the size of the list to be n we get a lower bound of $2n/(n+1)$ for the competitive ratio of any deterministic algorithm, and for any randomized algorithm against an adaptive off-line adversary.

Theorem 4.4 *If A is c-competitive against an adaptive on-line adversary, then $c \geq 2$.*

Proof Suppose that A is an on-line algorithm. Consider $\hat{A}$, an adaptive on-line adversary, which behaves as follows. $\hat{A}$ first simulates A on all possible choices of A's random bits, and for each such choice of random bits, creates a request sequence of length m such that each request is to the last item in A's list. Consider the collection

of all request sequences constructed in this way. The choice of A's random bits induces a probability distribution on these (indeed, all) sequences of length m. That is, the probability of a sequence is the probability that $\hat{A}$ will generate that sequence when it runs A and uses the strategy of always accessing the last item in A's list. $\hat{A}$ can now compute the quantities e_i of Lemma 4.1. Now, if $\hat{A}$ were to run A and construct a sequence such that each request is to the last item in A's list then A would surely pay n per access. However, by Lemma 4.1, $\hat{A}$ can arrange to pay as close as desired to $(n+1)/2$ per access on average, using the simple static algorithm. Therefore A cannot be better than $2n/(n+1)$-competitive. Since this must hold for all n, A cannot be better than 2-competitive. ∎

Theorem 4.4 shows that no randomized on-line algorithm can beat the deterministic lower bound against an adaptive on-line algorithm. It is known that the analogous theorem to Theorem 4.4 holds for other types of on-line situations such as the k-server problem, but it is not known to hold for the general request-answer games of [1]. The relationship between on-line adaptive and off-line adaptive adversaries in more general settings is an interesting open problem.

5 Lower Bounds Against Oblivious Adversaries

In this section we discuss a technique for deriving lower bounds for randomized on-line algorithms against oblivious adversaries. In this section, when we refer to competitive ratios, we mean competitive ratio against an oblivious adversary.

Karp and Raghavan [private communication, 1990] use the following strategy: find an off-line algorithm with a small average cost per request assuming that the requests are drawn uniformly at random from $\{1, 2, \ldots, n\}$.

Lemma 5.1 *Let $r_1, r_2, \ldots$ be an infinite sequence of requests each drawn uniformly and independently from $\{1, 2, \ldots, n\}$. Suppose there is an off-line algorithm, A, such that*

$$\lim_{m \to \infty} (1/m) \mathrm{E}\left[\mathrm{Cost}(A, \langle r_1, r_2, \ldots, r_m \rangle)\right] = d.$$

Then no on-line algorithm achieves a competitive ratio smaller than $(n+1)/2d$.

Proof If the requests are generated randomly, then the expected cost per access for any on-line algorithm is $(n+1)/2$, since at each request, no matter what the list may look like, each item has equal probability of being chosen. Since there is an off-line algorithm which has expected cost per request tending to d, no on-line algorithm can achieve a better competitive ratio than $((n+1)/2)/d = (n+1)/2d$. ∎

So, the strategy now is to come up with an off-line algorithm which has a small cost per request in the limit, given that the requests are generated randomly. To illustrate the ideas, we present an unpublished result for 2 item lists due to Karp.

Theorem 5.2 (Karp) *No on-line algorithm can achieve a smaller competitive ratio than 9/8 for two item lists.*

Proof Consider the following off-line algorithm for maintaining a two item list: If the current request is to the front item, do nothing. If the current request is to the back item, move that item to the front (via free exchanges) if and only if it is also requested next.

We can model the cost of successive requests by a markov chain with two states: state 1 corresponds to cost = 1 (that is, the current request is to the front item), and state 2 corresponds to cost = 2 (the current request is to the back item). The transition matrix for this chain is

$$\begin{pmatrix} 1/2 & 1 \\ 1/2 & 0 \end{pmatrix},$$

and the stationary distribution is 2/3 for state 1 and 1/3 for state 2. Therefore, the cost per access tends to $1(2/3) + 2(1/3) = 4/3$. So, we can apply Lemma 5.1 to show that no algorithm is better than 9/8-competitive. ∎

Raghavan [private communication, 1990] designed a similar one-lookahead algorithm for three item lists; using this algorithm, he showed a lower bound of 1.18. In order to improve the bound we must look at a larger list, construct a more complicated off-line algorithm, and compute the average cost per request for that algorithm. Fix the size of the list, n, and a *lookahead* number, k. Suppose we have an off-line algorithm that bases its decision on only the next $k+1$ requests. Following Raghavan's approach, we can model the behavior of such an algorithm for randomly generated request sequences as follows. Consider a markov chain whose states are k-tuples of request positions. State $\langle r_1, r_2, \ldots, r_k \rangle$ corresponds to the situation where the current *positions* of the next k requests are, respectively, $r_1, r_2, \ldots, r_k$. The transitions are conditioned on the current position of the $(k+1)$st request. If we know the algorithm, we can construct the transition matrix for the chain.

Each state transition has a cost (the cost of any paid exchanges made just prior to the access plus the cost of the access), so for each state we can compute an expected cost. Let the vector of these expected costs be $\vec{c}$. Suppose the chain has stationary distribution $\vec{\pi}$. Then the expected cost per access is $\vec{c} \cdot \vec{\pi}$, if the chain is in steady state. In any event, as long as the chain converges to steady state, the cost per request tends to $\vec{c} \cdot \vec{\pi}$ as the length of the request sequence increases.

For $n \geq 3$ there is no obvious strategy for a k-lookahead algorithm, since it seems no bounded lookahead algorithm can be optimum. Furthermore, the size of the transition matrix is n^k, so for values of n and k even a little bigger than three it is infeasible to try different strategies by hand and compute the steady state vector symbolically. Thus there are two problems: determining a good way to find off-line algorithms and generate transition rules for large lists, and finding the steady state distribution of the resultant transition matrix.

Our approach is to write a program that generates the entries of the matrix corresponding to a particular class of good off-line algorithms, MARKOV(n,k), and then compute the steady state vector numerically. Suppose we have a program OFF(n,k) that, given a sequence of exactly $k+1$ accesses to an n item list, generates a sequence of moves to service that request sequence. MARKOV(n,k) works as follows: start with the next k requests; look at the $(k+1)^{st}$ request; service the next request in the same way that OFF(n,k) would service that request given the same sequence of $k+1$ requests on the same list. Given an implementation of OFF(n,k), the entries of the transition matrix and cost vector for MARKOV(n,k) are easily filled in. If the current state is $\langle r_1, r_2, \ldots, r_k \rangle$ (that is, if the next k requests are to the items in positions $r_1, r_2, \ldots, , r_k$), and the next request is to the item in position r_{k+1}, transit to whatever state corresponds to the list that results from OFF(n,k) servicing request r_1 in the sequence $\langle r_1, r_2, \ldots, r_{k+1} \rangle$, assuming its list is $1, 2, \ldots, n$. The probability of that transition is $1/n$, since the request sequence is uniformly random. The cost of the state is $1/n$ times the sum over all choices of r_{k+1} of the cost of the servicing the first request in $\langle r_1, r_2, \ldots, r_{k+1} \rangle$. Thus to generate the matrix we must generate all possible sequences of $k+1$ requests on n items and run OFF(n,k) on each of them.

Any choice of an off-line algorithm gives a valid MARKOV(n,k) algorithm and transition matrix, but some choices are better than others. Our best results were achieved using a program to compute an optimum off-line algorithm for the $k+1$ requests that has the following properties:

1. It only does paid exchanges.

2. It services a request to item x by choosing some subset of the items preceding x in the list and moving them in an order-preserving way to immediately after x. Then it pays for the access to x and goes on to the next request.

3. Whenever an item is requested twice in a row that item is moved to the front on its first access. This means that MARKOV(n,k) has the same property and ensures that the markov chain has at most one stationary distribution (see Lemma 5.3).

4. Among optimum algorithms that have the above three properties, our algorithm services the first request with the lowest cost. This tends to make MARKOV(n,k)'s cost per access smaller, so we get a better lower bound.

In ([9]) we prove that there is always an optimum algorithm with the first three properties, and discuss implementation details. There is no known way to compute the optimum algorithm for a given request sequence that is polynomial in both n and k, however, so the time and space to generate MARKOV(n,k) grow exponentially.

Lemma 5.3 *For any n and k, the markov chain corresponding to MARKOV(n,k) is irreducible. That is, for any two states, the probability of transiting from one to the other in some finite time is positive.*

n	k	Lower Bound
3	10	1.1998
4	8	1.2467
5	7	1.2728
6	5	1.268

Table 1: Lower bound results

Proof Suppose we want to get to state $\langle r_1, r_2, \ldots, r_k \rangle$. Consider MARKOV$(n,k)$'s action on request sequence $\langle n, n, n-1, n-1, \ldots, 2, 2, 1, 1 \rangle$. MARKOV$(n,k)$ will move each item to the front, since each item is requested twice in a row. After that, MARKOV(n,k)'s list must be $1, 2, \ldots, n$, so if the next k requests are $\langle r_1, r_2, \ldots, r_k \rangle$, MARKOV$(n,k)$ will be in state $\langle r_1, r_2, \ldots, r_k \rangle$. ∎

By standard probability theory, Lemma 5.3 implies that the steady state distribution is unique, and it is given by the (unique) eigenvector of the transition matrix corresponding to the eigenvalue 1. We compute this eigenvector from the matrix using the power method [4]. Table 1 shows the best results we have obtained for $n = 3, 4, 5$ and 6. In all cases, the number of iterations in the power method necessary to get the distance between successive iterates less than 10^{-7} was no more than 30.

As shown in the table, the largest lower bound we have been able to compute is approximately 1.27. We were unable to achieve higher results due to limitations on computational resources. From various simulations using non-optimum off-line algorithms we believe the true lower bound to be at least 1.4.

6 Open Problems and Remarks

We have given an algorithm that is 1.75-competitive against an oblivious adversary, and constructed algorithms with somewhat better competitive ratios. We have shown that no algorithm can be better than 1.27-competitive against such an adversary. This, of course, leaves a large gap, and a very interesting open problem. Furthermore, we do not know of an instance in which the upper bound for BIT is tight.

The results for BIT can be extended to a closely related list update model of interest in which an access to the i^{th} item in the list has cost $i-1$ rather than i. Any algorithm that costs C on some sequence σ of accesses in the i cost model will cost $C-m$ in the $i-1$ model, where $m = |\sigma|$. Applying Theorem 3.1, we have that in the $i-1$ model $\text{Cost}(BIT, \sigma) + m \leq 1.75(\text{Cost}(OPT, \sigma) + m) - 3m/4$, which implies that $\text{Cost}(BIT, \sigma) \leq 1.75\text{Cost}(OPT, \sigma)$. Our results can also be generalized to a convex cost model, in which an access to the i^{th} item has cost $f(i)$ and a paid exchange of the i^{th} item with its successor has cost $f(i+1) - f(i)$, for some convex function f.

Sleator has produced a $\sqrt{3}$-competitive algorithm, using a variation of the idea behind the COUNTER algorithms. In Sleator's algorithm, each item has a counter which can be 1, 2, or 3. When an accessed item is in state 1 it moves to the front. If

it is not in state 1 it does not move. The value of the counter is changed as follows. If the counter is 3 or 2, it is decremented. If the counter is 1 then after moving to front, the counter is set to 3 with probability $(-1+\sqrt{3})/2$ and to 2 with probability $(3-\sqrt{3})/2$. The analysis is the same as for the COUNTER algorithms. Notice, however, that the number of random bits used by Sleator's algorithm is $\Omega(m)$, where m is the length of the request sequence. It is possible to choose s and S so that the competitive factor of COUNTER(s, S) guaranteed by Theorem 3.4 is as close to $\sqrt{3}$ as desired (though s must tend to infinity). Although such a COUNTER algorithm potentially has a very large finite state machine, the total number of random bits needed is still $O(n)$.

In addition, we have shown that no algorithm can hope to be better than 2-competitive against an adaptive on-line adversary. This can obviously be achieved with the move-to-front. Thus both adaptive on-line and adaptive off-line adversaries are equivalent in their power against list-update algorithms.

Our results complement those found in the other well-studied application of competitive analysis, page caching. In this application, randomization helps against oblivious adversaries, and is no use against either kind of adaptive adversaries. Furthermore, both kinds of adaptive adversaries are equivalent in power against page caching algorithms. In fact, there is no known application in which randomized algorithms against adaptive on-line adversaries fare any better than the best deterministic bounds, i.e. in which on-line adversaries are any weaker than off-line adversaries. Our results in this paper give evidence for the conjecture that for a large class of applications, adaptive on-line equals adaptive off-line. It is a very interesting open problem to prove this conjecture or find an example that delimits the class.

7 Acknowledgements

We thank Prabhakar Raghavan for introducing us to this problem. We also thank Richard Beigel, Sandy Irani, Danny Sleator, and Neal Young for useful discussion.

References

[1] S. Ben-David, A. Borodin, R. M. Karp, G. Tárdos, and A. Wigderson. On the power of randomization in on-line algorithms. In *Proc. 20th ACM Symposium on Theory of Computing*, pages 379–386, May 1990.

[2] J. L. Bentley and L. McGeoch. Worst-case analysis of self-organizing sequential search heuristics. In *Proceedings of 20th Allerton Conference on Communication, Control, and Computing*, pages 452–461, Urbana-Champaign, October 1982. University of Illinois.

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

[4] S. D. Conte and C. de Boor. *Elementary Numerical Analysis, An Algorithmic Approach.* McGraw-Hill, third edition, 1980.

[5] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. D. Sleator, and N. Young. On competitive algorithms for paging problems. To appear in *Journal of Algorithms*, 1988.

[6] G. H. Hardy, J. E. Littlewood, and G. Pòlya. *Inequalities.* Cambridge University Press, Cambridge, second edition, 1934.

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

[8] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. Research Report RC 15622 (No. 69444), IBM T. J. Watson Reseach Center, 1990.

[9] N. Reingold and J. Westbrook. Optimal off-line algorithms for the list update problem. Technical Report YALEU/DCS/TR-805, Yale University, 1990.

[10] R. Rivest. On self-organizing sequential search heuristics. *Communications of the Association for Computing Machinery*, 19(2):63–67, February 1976.

[11] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the Association for Computing Machinery*, 28(2):202–208, 1985.