A Combined BIT and TIMESTAMP Algorithm for the List Update Problem

Information Processing Letters **56** (1995) 135–139

Susanne Albers, Bernhard von Stengel, Ralph Werchner

International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704, USA
Email: {albers, stengel, werchner}@icsi.berkeley.edu

Abstract. We present a randomized on-line algorithm for the list update problem which achieves a competitive factor of 1.6, the best known so far. The algorithm makes an initial random choice between two known algorithms that have different worst-case request sequences. The first is the BIT algorithm that, for each item in the list, alternates between moving it to the front of the list and leaving it at its place after it has been requested. The second is a TIMESTAMP algorithm that moves an item in front of less often requested items within the list.

Keywords. On-line algorithms, analysis of algorithms, competitive analysis, list-update.

1. Description of the algorithm

The *list update problem* is one of the first on-line problems that have been studied with respect to competitiveness (see [5] and references). The problem is to maintain an unsorted list of items so that access costs are kept small. An initial list of items is given. A sequence of *requests* must be served in that order. A request specifies an item in the list. The request is served by accessing the item, incurring a cost equal to the position of the item in the current list. In order to reduce the cost of future requests, an item may be moved free of charge further to the front after it has been requested. This is called a *free exchange*. Any other exchange of two consecutive items in the list incurs cost one and is called a *paid exchange*. The goal is to serve the request sequence so that the total cost is as small as possible.

An *on-line* algorithm has to serve requests without knowledge of future requests. An optimal *off-line* algorithm knows the entire sequence σ of requested items in advance and can serve it with minimum cost $OPT(\sigma)$. We are interested in the *competitiveness* of an on-line algorithm. Let $A(\sigma)$ be the cost incurred by the on-line algorithm A for serving the sequence σ. Then the algorithm is called c-competitive if there is a constant b so that $A(\sigma) \leq c \cdot OPT(\sigma) + b$ for all request sequences σ. The smallest c with this property is called the *competitive factor* of the algorithm.

The well-known MOVE-TO-FRONT rule is 2-competitive, which is optimal for deterministic algorithms [5, 6]. The performance of a *randomized* on-line algorithm A can be better if it is evaluated against the *oblivious adversary* [2]. The oblivious adversary specifies a request sequence σ in advance and is not allowed to see the random choices made by the on-line algorithm A. Let $E[A(\sigma)]$ denote the corresponding expected cost. The algorithm is called c-competitive if there is a constant b so that $E[A(\sigma)] \leq c \cdot OPT(\sigma) + b$ for all request sequences σ. Against this adversary, which is considered in this paper, the optimal competitive factor of a randomized on-line algorithm for the list update problem is not yet known.

The cost of accessing the ith item in the list is i. However, to simplify our analysis, we assume this cost to be $i - 1$ instead. Clearly, a c-competitive on-line algorithm for this $(i - 1)$-cost model is also c-competitive in the i-model. With either cost model, it is known that no randomized on-line algorithm for the list update problem can be better than 1.5-competitive [7].

We will combine two on-line algorithms for the list update problem that store with each item some information about past requests. Both algorithms use only free exchanges.

The first algorithm is the 1.75-competitive BIT algorithm due to Reingold, Westbrook, and Sleator [5]. The algorithm maintains a bit for each item in the list. Initially, the bit is set at random to 0 or 1 with equal probability so that the bits of the items are pairwise independent.

Algorithm BIT. *Each time an item is requested, its bit is complemented. When the value of the bit changes to 1, the requested item is moved to the front of the list. Otherwise the position of the item remains unchanged.*

The second algorithm is an instance of the TIMESTAMP algorithm recently introduced by Albers [1]. This algorithm maintains for each item the last two times it has been requested. An item is treated in one of two ways (which can be determined once at the beginning by a random experiment, so that the algorithm is *barely random*, that is, it uses only a bounded number of random bits independent of the number of requests [5]). With probability p, the item is moved to the front of the list after it has been requested. With probability $1 - p$, it is treated in a

different way. The TIMESTAMP algorithm with parameter p has a competitiveness of $\max\{2-p, 1+p(2-p)\}$. The optimal choice of p gives a ϕ-competitive algorithm, where $\phi = (1+\sqrt{5})/2 \approx 1.62$ is the Golden Ratio.

We use the TIMESTAMP algorithm with parameter $p = 0$, so that it is deterministic. The resulting 2-competitive algorithm can be formulated as follows.

Algorithm TS. *After each request, the accessed item x is inserted immediately in front of the first item y that precedes x in the list and was requested at most once since the last request to x. If there is no such item y or if x is requested for the first time, then the position of x remains unchanged.*

Our new algorithm is a combination of these two algorithms.

Algorithm COMB. *With probability $4/5$ the algorithm serves a request sequence using BIT, and with probability $1/5$ it serves the sequence using TS.*

Theorem 1. *The on-line algorithm COMB is 1.6-competitive.*

In the following, we will prove Theorem 1 using a well-known technique [3, 4] of analyzing separately the movement of any pair of items in the list. The algorithms *BIT* and *TS* permit such a pairwise analysis.

2. Projection on pairs of items

Our goal is to look only at two items at a time when we consider a request sequence, the list maintained by the on-line algorithm, and the cost of the off-line algorithm. Let σ be a sequence of m requests, and let $\sigma(t)$ be the item requested at time t for $t = 1, \ldots, m$. Let L be the set of items of the list. Consider any deterministic algorithm A that processes σ. At time t, requesting $\sigma(t)$ incurs a cost that depends on the current list maintained by A. This cost can be represented as the sum

$$\sum_{x \in L} A(t, x)$$

where $A(t, x)$ is equal to one if item x precedes $\sigma(t)$ in the list at time t, and zero otherwise. The cost $A(\sigma)$ of serving the entire sequence σ has then the following form, using $A(t, x) = 0$ for $x = \sigma(t)$:

$$\begin{aligned} A(\sigma) &= \sum_{t=1,\ldots,m} \sum_{x \in L} A(t, x) \\ &= \sum_{x \in L} \sum_{t=1,\ldots,m} A(t, x) \\ &= \sum_{x \in L} \sum_{y \in L} \sum_{t\,:\,\sigma(t)=y} A(t, x) \\ &= \sum_{\{x,y\} \subseteq L\,:\,x \neq y} \sum_{t\,:\,\sigma(t) \in \{x,y\}} \Big(A(t, x) + A(t, y) \Big) . \end{aligned}$$

With the abbreviation

$$A_{xy}(\sigma) = \sum_{t\,:\,\sigma(t)\in\{x,y\}} \Big(A(t,x) + A(t,y)\Big)\,, \tag{1}$$

we can write this as

$$A(\sigma) = \sum_{\{x,y\}\subseteq L\,:\,x\neq y} A_{xy}(\sigma)\,. \tag{2}$$

Let σ_{xy} be the request sequence σ with all items other than x or y deleted. Only these requests are considered in (1). In the sum there, $A(t,x) + A(t,y)$ is the cost of accessing $\sigma(t)$ in the two-element list that consists of the items x and y in the same relative order as in the full list. In that way, the term $A_{xy}(\sigma)$ denotes the cost of the algorithm 'projected' to the unordered pair $\{x,y\}$ of items.

The algorithms *BIT* and *TS* are compatible with the projection on pairs. That is, when these algorithms serve a request sequence σ, then at any time the relative order of two items x and y in the list depends only on the projected request sequence σ_{xy} and the initial order of x and y. This is obvious for the algorithm *BIT* which moves an item independently of any other item. For the algorithm *TS*, this follows from the following lemma, applied to the request sequence σ or any prefix of it.

Lemma 2. *In the list obtained after algorithm TS has served the request sequence σ, item x precedes item y if and only if the sequence σ_{xy} terminates in the subsequence xx, xyx, or xxy, or if x preceded y initially and y was requested at most once in σ.*

Proof. Suppose σ_{xy} terminates in xx or xyx, and let y precede x in the list at the time of the last request to x. Then y is among the items that have been requested at most once since the preceding request to x. Since x is inserted in front of the first of such items, x precedes y in the final list.

Let σ_{xy} terminate in the subsequence xxy, and let t_1, t_2, and t_3 be the times of these last three requests to x or y. After the request to x at time t_2, item x is moved somewhere in front of y. Suppose that after the request to y at time t_3, item y is, contrary to our claim, moved somewhere in front of x, and suppose further that y is the *first* of the items in σ that has not been requested between t_1 and t_2 and is requested after t_2 and then moved in front of x. Let t_0 be the time of the preceding request to y (item y must be requested at least twice to be moved), where $t_0 < t_1$. Then y is inserted immediately in front of an item z that has been requested at most once between t_0 and t_3, so $z \neq x$, and z is in front of x at time t_3. If z was requested before t_2, then σ_{xz} ends in xx or xzx, where we have shown that x is in front of z after t_2. So z is an item that has not been requested between t_1 and t_2 and is requested after t_2 and then moved in front of x, but before the request to y at time t_3, contradicting our assumption that y is the first of such items. Thus x precedes y in the final list as claimed.

If σ_{xy} terminates in one of the subsequences yy, yxy, or yyx, then by the same argument with x and y interchanged, y precedes x in the final list.

The only remaining cases are when both x and y are requested at most once in σ. Then neither item is moved, so their relative order is as in the initial list. $\square$

By Lemma 2, the relative order of any two items x and y in the list when TS serves σ is the same as when TS serves σ_{xy} on the two-element list consisting of x and y. In other words, $TS_{xy}(\sigma) = TS(\sigma_{xy})$, where $TS(\sigma_{xy})$ denotes the cost of TS serving σ_{xy} on the two-element list (with x and y in the same initial order as in the long list). Similarly, the projected cost of the algorithm BIT fulfills $BIT_{xy}(\sigma) = BIT(\sigma_{xy})$. Note that this cost is a random variable.

For the optimal off-line algorithm OPT, we work with the inequality

$$OPT_{xy}(\sigma) \geq \overline{OPT}(\sigma_{xy}), \tag{3}$$

which states that the projected cost of OPT processing σ is at least as high as the optimal off-line cost $\overline{OPT}(\sigma_{xy})$ of serving σ_{xy} on the two-element list. An optimal off-line algorithm $\overline{OPT}$ for only two items can be easily specified. However, (3) may not always hold with equality. In that case, the moves of $\overline{OPT}$ for all pairs of items cannot be combined to yield an algorithm for the entire list. The different notation $\overline{OPT}$ emphasizes that this algorithm may perform better than the projection of OPT serving requests on a longer list.

A randomized algorithm can be regarded as a probability distribution on deterministic algorithms A. Then, (2) carries over to expected values. For the expected cost of our on-line algorithm $COMB$ we will prove in Section 3 the inequality

$$E[COMB_{xy}(\sigma)] \leq 1.6 \cdot \overline{OPT}(\sigma_{xy}) \tag{4}$$

for all pairs $\{x, y\}$ of items. By the preceding discussion, this implies $E[COMB(\sigma)] \leq 1.6 \cdot OPT(\sigma)$ and thus shows Theorem 1.

3. Competitiveness of the algorithm

As shown in the previous section, the competitiveness of the algorithm $COMB$ can be analyzed considering only request sequences σ_{xy} to the items x and y in a two-element list. We partition σ_{xy} into subsequences, each of which is terminated by two consecutive requests to the same item. Assuming that x precedes y in the initial list, the first subsequence is of the form $x^l yy$, $x^l (yx)^k yy$, or $x^l (yx)^k x$ for some $l \geq 0$ and $k \geq 1$. If that subsequence terminates in xx, the next subsequence is of the same form. If it terminates in yy, we consider next the subsequence of one of these forms with x and y interchanged. Continuing in this manner, σ is partitioned uniquely. We can assume that the last subsequence is also of this form by appending the requests yy to σ_{xy}, which affects costs negligibly.

It suffices to prove (4) for each subsequence. The cost for such a subsequence of σ_{xy} is the same as when the subsequence is served by itself, for the following reason: Whenever an item has been requested twice in a row, it is moved to the front by *BIT* and *TS*. For $\overline{OPT}$, we can assume the same behavior, because it is optimal to move the item, say x, to the front after the first of two or more consecutive requests to x. Thus, after a subsequence of σ_{xy} ending with xx has been served by *BIT*, *TS*, or $\overline{OPT}$, these algorithms start on the next subsequence with item x at the front of the list. When algorithm *BIT* is used, the bits of some items may have changed, but the expected cost is not affected; algorithm *TS* treats any request to y after the requests xx as if y is requested for the first time.

The cost for serving a subsequence varies with the algorithm. We first prove a lemma for the *BIT* algorithm.

Lemma 3. *Suppose that BIT has served the request sequence xyx, or the sequence yx on a list where initially x preceded y. Then x is in front of y with probability 3/4.*

Proof. We show that after *BIT* has served either sequence, item y is in front of x if and only if the bit of x is 0 and the bit of y is 1: Namely, if the bit of x was set to 1 at the last request to x, then x was moved to the front. Otherwise, x's bit is 0, so the bit was set to 1 at the preceding request to x (in the sequence xyx) and x is front of y at the time of the request to y (which holds by assumption for the sequence yx). Thus, y's bit must have been set to 1 after the request to y to move y in front. The bits of both items are independent, so y is in front of x with probability 1/4. □

Lemma 4. *In the initial list of two items, let x be in front of y. The following table describes the expected cost for serving the indicated request sequences, where $l \geq 0$ and $k \geq 1$, by the algorithms BIT, TS, and $\overline{OPT}$.*

request sequence	*BIT*	*TS*	$\overline{OPT}$
$x^l yy$	$\frac{3}{2}$	2	1
$x^l (yx)^k yy$	$\frac{3}{2}k + 1$	$2k$	$k + 1$
$x^l (yx)^k x$	$\frac{3}{2}k + \frac{1}{4}$	$2k - 1$	k

Proof. The initial l requests to x incur no cost for any of the algorithms. Consider the request sequence $x^l yy$. Since x precedes y before the first request to y, the cost of serving that request is 1. After that request, algorithm *BIT* moves item y to the front with probability 1/2 so that the expected cost for the service by *BIT* is 3/2. Algorithm *TS* incurs cost 1 at both requests to y by Lemma 2. Clearly, the optimal off-line algorithm $\overline{OPT}$ moves y to the front after the first request to y.

The sequence $x^l(yx)^k yy$ is served by *BIT* as follows: The first subsequence yx incurs expected cost 3/2 since the first request costs 1, after which y is moved to the front with probability 1/2, so the request to x has expected cost 1/2. Any further request to y or x in the subsequence $(yx)^k$ incurs expected cost 3/4 by Lemma 3 (and Lemma 3 with x and y interchanged). Lemma 3 also shows that the final two requests to y have expected cost 3/4 and 1/4, respectively. Thus, the *BIT* algorithm serves $x^l(yx)^k yy$ with expected cost $\frac{3}{2}k + \frac{3}{4} + \frac{1}{4}$. By the same reasoning, that cost for the sequence $x^l(yx)^k x$ is $\frac{3}{2}k + \frac{1}{4}$.

When algorithm *TS* serves the sequence $x^l(yx)^k yy$, then the first two requests of the form yx incur costs 1 and 0, respectively, since y is left behind x after the first request to y. All subsequently requested items are moved to the front of the list by Lemma 2. The resulting costs are therefore $2k$ (note $k \geq 1$). Similarly, *TS* serves $x^l(yx)^k x$ with cost $2k - 1$.

The optimal off-line cost for serving the sequence $x^l(yx)^k yy$ is $k + 1$ since for each of the k pairs yx of requests, at least one has cost 1, and an extra cost unit is caused by the final double request to y. It is optimal to move y to the front at any time before the last request to y. The optimal off-line cost for serving the sequence $x^l(yx)^k x$ is k. It is optimal to leave x always at the front of the list. □

The performance of algorithm *COMB*, which selects *BIT* with probability 4/5 and *TS* with probability 1/5, follows from Lemma 4. *COMB* serves the request sequence $x^l yy$ with expected cost 1.6, the sequence $x^l(yx)^k yy$ with cost $1.6k + 0.8$, and the sequence $x^l(yx)^k x$ with cost $1.6k$. In each case, this is at most 1.6 times the cost of $\overline{OPT}$. This proves (4) and thus Theorem 1.

The probabilities for deciding between *BIT* and *TS* are optimal: The critical sequences are $x^l yy$ and $x^l(yx)^k x$ (for $x^l(yx)^k yy$ *COMB* performs better), were the simplest cases are yy with expected cost 1.5 for *BIT* and 2 for *TS*, and yxx with expected cost 1.75 for *BIT* and 1 for *TS*. If a randomizing adversary chooses yy with probability 3/5 and yxx with probability 2/5, then both *BIT* and *TS* have expected cost 1.6, or 1.6 times the cost of $\overline{OPT}$. Thus, by Yao's Theorem [8] (or a simple direct argument), no randomized combination of *BIT* and *TS* can have cost less than 1.6 on both sequences (and their repetitions in longer sequences).

4. Conclusions

We have presented a simple randomized on-line algorithm for the list update problem that has a competitive factor of 1.6. The best known lower bound for that factor is 1.5 [7]. The remaining gap is small, but the best possible competitive factor remains to be determined. We have found a 1.5-competitive algorithm (not presented in this paper) for serving requests on a list of up to four items, but it cannot be generalized to longer lists.

References

[1] S. Albers, Improved randomized on-line algorithms for the list update problem, *Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms* (1995) 412–419.

[2] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson, On the power of randomization in on-line algorithms, *Algorithmica* **11** (1994) 2–14.

[3] J. L. Bentley and C. C. McGeoch, Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM* **28** (1985) 404–411.

[4] S. Irani, Two results on the list update problem, *Information Processing Letters* **38** (1991) 301–306.

[5] N. Reingold, J. Westbrook, and D. D. Sleator, Randomized competitive algorithms for the list update problem, *Algorithmica* **11** (1994) 15–32.

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

[7] B. Teia, A lower bound for randomized list update algorithms, *Information Processing Letters* **47** (1993) 5–9.

[8] A. C. Yao, Probabilistic computations: Towards a unified measure of complexity, *Proc. 18th Annual IEEE Symposium on Foundations of Computer Science* (1977) 222–227.