A Combined BIT and TIMESTAMP Algorithm for the List Update Problem

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. A simple randomized on-line algorithm for the list update problem is presented that 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, linear lists, 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 a so that $A(\sigma) \leq c \cdot OPT(\sigma) + a$ for all request sequences σ. The smallest c with this property is called the *competitive factor* of the algorithm.

The best possible deterministic algorithms for the list update problem are 2-competitive [4, 6]. The optimal competitive factor of a *randomized* on-line algorithm is not yet known. We evaluate its performance 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. Against the oblivious adversary, the algorithm is c-competitive if there is a constant a so that $E[A(\sigma)] \leq c \cdot OPT(\sigma) + a$ for all request sequences σ.

Usually, the cost of accessing the ith item in the list is i. For simplicity, we assume that 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 original 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 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].Depending on a parameter p in $[0, 1]$, this algorithm achieves 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. The TIMESTAMP 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* as defined in [5]). With probability p, the item is moved to the front of the list after is has been requested. With probability $1-p$, it is treated in a different way. 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, 5] 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 order of the full list maintained by A. 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 can be told from 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. Then at the last request to x, item 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. Then y is inserted immediately in front of an item z that has been requested at most once since the preceding request to y, which took place before t_1. So z precedes x at time t_3, but then clearly z must have been requested twice since t_1, a contradiction. 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 always in the same initial order as in the long list). Similarly, the projected cost of the algorithm *BIT* fulfills $BITxy(\sigma) = BIT(\sigma_{xy})$. 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, but the corresponding moves for all pairs of items may not be implementable on a longer 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 for all pairs $\{x, y\}$ of items the inequality

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

which by the preceding discussion 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. It suffices to analyze only such subsequences, for the following reason: Whenever an item has been requested twice in a row, we can assume that it is moved to the front by BIT, TS, and $\overline{OPT}$. This holds always for BIT and TS. It also optimal for $\overline{OPT}$ to move the item, say x, to the front after the first of two or more consecutive requests to x, since then the cost of serving these requests is 1, plus a possible additional cost 1 for the next request to y; if x were kept at the end of the list, the cost would be at least 2. Finally, we disregard the final subsequence of σ_{xy}, which may not end in a double request, since it is irrelevant for an asymptotic analysis with very long request sequences.

Thus, after a subsequence of σ_{xy} ending with xx has been served by BIT, TS, or $\overline{OPT}$, item x is at the front of the list (the same holds for y instead of x). Furthermore, we can treat the remainder of the sequence as a new request sequence, served on an initial list with x in front of y: 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.

In the initial two-element list, we assume x precedes y. Consecutive requests to x at the beginning of the request sequence incur no cost. After y has been requested for the first time, we consider the requests until the first double request xx or yy. The resulting sequence is of one of three possible forms that we study separately: $x^l yy$, $x^l(yx)^k yy$, or $x^l(yx)^k x$ for some $l \geq 0$ and $k \geq 1$. The cost for serving these sequences varies with the algorithm.

Lemma 3. *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. Consider the second and any further request to y which is preceded by three requests of the form xyx at times t_1, t_2, and t_3, say (or by two requests yx at times t_2 and t_3 where at time t_2 item x is with certainty in front of y; then the following argument applies as well). We claim that at that second request to y (at time t_4, say), y is at the front of the list 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 at t_3, then x was moved to the front. If x's bit is 0 at time t_4, then x's bit was set to 1 after the service of x at time t_1 so that x is with certainty in front of y at time t_2. Thus, y's bit must have been set to 1 after the request to y at time t_2 to move y in front. This shows the claim. The bits of both items are independent, so y is in front at time t_4 with probability 1/4 and the expected cost of serving y is 3/4. By the same argument, all but the first two requests to y or x in the subsequence $(yx)^k$ incur expected cost 3/4. Of the final two requests yy, the first request to y also has expected cost 3/4. The other is the last request of a subsequence $yxyy$; it is easy to see that at that time, y is *not* in front of x if and only if y's bit is 0 and x's bit is 1, which happens with probability 1/4. 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 3. *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 $1.6k + 1.6$. 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 the sequence 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 obvious open question is: what is the best possible competitive factor?

Our algorithm uses two known algorithms that already have good competitive factors. We have used the fact that the worst-case request sequences for these algorithms are different. In a similar way, it is possible to construct other 1.6-competitive algorithms. For example, one can use the original TIMESTAMP algorithm [1] with different parameters p, determined by a random experiment: With probability 1/10, TIMESTAMP is used with $p = 0$ (corresponding to our algorithm *TS*), with probability 1/15, it is used with $p = 1$ (yielding the deterministic MOVE-TO-FRONT rule), and with probability 5/6, it is used with $p = 2/5$. This means that an item is always moved to the front with probability 2/5, otherwise treated essentially as in algorithm *TS* (see [1] for a full description of TIMESTAMP when p is not 0 or 1). However, with probability 1/6 *all* items are treated in the same manner. This correlation reduces the competitive factor of the algorithm from the Golden Ratio (about 1.62) to 1.6. The analysis of this algorithm is similar to Lemma 3. However, *COMB* is simpler and uses with high probability the easily implementable *BIT* algorithm.

It may be that the optimal competitive factor is indeed 1.5. There is a 1.5-competitive algorithm for serving requests on a list of up to four items. That algorithm is based two-dimensional partial orders, but it is beyond this article to describe; furthermore, the algorithm cannot be extended to longer lists. Its performance can also be compared against $\overline{OPT}$, assuming that the optimal off-line algorithm projects to pairs. It is conceivable that for longer lists, $\overline{OPT}$ does not suffice to describe the performance of OPT (that is, the inequality in (3) is strict) for certain critical request sequences. In that case, the optimal competitive factor must be analyzed by other tools than the projection to pairs of items, which has so simplified our analysis.

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

[4] R. Karp and P. Raghavan, Personal communication (1990), cited in [5].

[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.