On Lookahead in the List Update Problem

Rahul Simha

Department of Computer Science, The College of William & Mary, Williamsburg, VA 23185

Amitava Majumdar

Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, IL 62901

Communicated by D.Gries
Received 25 August 1993
Revised 20 December 1993

Abstract

Simha, R. and A. Majumdar, On Lookahead in the List Update Problem. Information Processing Letters.

We study a problem in the area of self-adjusting data structures: algorithms for the singly-linked list for the case in which a sequence of operations is to be performed on the list and the entire sequence is known beforehand. Whereas past approaches assume that the cost of lookahead is zero, we explicitly take into account this cost and allow for different list access and lookahead costs, parametrically specified by a ratio α. Our main result is that the well-known Move-to-Front algorithm is 2α-competitive with an algorithm that scans the entire sequence of operations. Simulation results point to a remarkably counter-intuitive observation: the optimal strategy is to either scan the entire sequence of operations or not scan it at all.

Keywords: Data structures, algorithms, analysis of algorithms.

1 Introduction

Recent work in the area of data structures has brought about several interesting and useful results on *dynamic* or *self-adjusting* data structures, many of these originating from the study of one of the simplest and most elegant data structures, the singly-linked list. The general framework of such investigations is simple to describe: given a data structure and a sequence of operations, and the opportunity to adjust the data structure after each operation, how should the data structure be adjusted to minimize the time taken to perform the operations? Typically, one compares *on-line* adjustment strategies (those that adjust based only on the current and past operations) with *off-line* ones (those that know the whole sequence of operations from the start). Known results in this area are particularly compelling when the cost of handling the sequence of operations is *amortized* over the sequence [14].

It is usually assumed that the off-line algorithm can *look ahead* into the future for free. This assumption has been motivated by the need to analyze on-line algorithms for applications in which the entire sequence of operations is not available beforehand, thus using the hypothetical performance of the off-line algorithm for comparison purposes only.

In this paper, we consider the list update problem when the entire sequence of operations is given at the start. Since our results hold for insertions and deletions with slight modification, we will assume that the sequence of operations is a string of accesses and we will refer to it as the *batch* of accesses. The linked list itself is called the *list*. Our

Correspondence to: Rahul Simha, Dept. of Computer Science, College of William & Mary, Williamsburg, VA 23185. Email: simha@cs.wm.edu

formulation differs from past ones in that we assume the batch is itself stored in a data structure comparable to a linked list — a natural assumption for batched queries. After all, if the batch is 'organized' in some informationally useful way, then *some* algorithm must perform this additional organizing. Thus, to make a fair comparison with algorithms that do not scan the batch, this overhead of scanning the batch should be part of the problem structure. In other words, to *look ahead*, a cost is incurred in manipulating the batch and this cost is roughly equal to that obtained in scanning the linked list itself. We will charge one unit of cost for each comparison made in the batch and α ($\alpha > 1$) units for each comparison made in the list. Our main result is that the well-known Move to Front (MTF) algorithm is 2α-*competitive* with an algorithm, which we call BATCH, that searches through the sequence of operations. We also consider algorithms that perform limited lookahead and present simulation evidence for the remarkably counter-intuitive result that the optimal lookahead should either be zero (MTF) or the size of the sequence (BATCH).

Our approach of considering a batch of accesses is motivated by applications in which operations are performed periodically after a group of operations are generated. One application is provided by data structure software written in the producer consumer style, where the producer places queries in a buffer ('batch') while the consumer (the program manipulating the data structure) is busy; when the consumer is free, it picks up the next 'batch' of queries. Often the code that searches through the data structure is written as a separate *procedure*, say as `Search (item)`; then, a call to `Search()` generates some function call overhead – we model this cost by the quantity α. Another application is found in packet-switched networks where routing software must perform the mapping between destination address and outgoing link for each packet in a burst of packets. Other applications are discussed in [17].

In Section 2, we review past related work. Section 3 contains our main result. We present some simulation results in Section 4 before concluding in Section 5.

2 Previous Work

Our work falls in the area of self-adjusting or dynamic data structures [15]. A survey for linked lists, in particular, is found in [8]. The bulk of the work on dynamic lists [1, 2, 4, 5, 6, 7, 11, 16, 13, 14]. has studied the well-known Move-to-Front (MTF), Transpose (TRANS) and Count heuristics. Probabilistic analyses of the problem appear in [2, 5, 6, 7, 16]. In [1] the notion of competitive analysis is introduced; it is shown that MTF and Count satisfy a pairwise independence property that enables comparison with the static optimal arrangement. Further analysis and extensions of these results appear in [14], in particular, the remarkable result that under certain conditions, MTF performs within a factor of two of the optimal off-line algorithm. Our work strengthens the support for MTF, by showing that MTF is 2α-competitive with BATCH, an algorithm that scans the batch for each access. The limited utility of such additional information is also the theme in [9], when the information is distributional and in the context of using counters.

Our proposed work comes closest to that of [17], in which the notion of simple independent queries is extended to groups of queries. In fact, we partially address an open problem mentioned in [17]: given groups of queries, when is it better to order the queries before searching the list? Note that in [17], a *set* of accesses is considered and the authors give a probabilistic analysis when a stream of randomly selected sets is generated. The problem of lookahead is not considered in [17]; also, our analysis is competitive [10] as opposed to probabilistic.

3 Main Result

Consider a singly-linked list L consisting of n elements, from a finite universe U. We are also given a batch $b = b_1 b_2 \ldots b_m$ of m accesses, for each of which one must determine whether $b_i \in L$ (the proof is easily modified to handle insertions and deletions). Note that if we take U to be an alphabet, then $b \in U^*$, the set of all strings over U. The *list cost* of an algorithm is the number of comparisons made in searching through the list. The *batch cost* is the number of comparisons made in scanning the batch; no batch cost is incurred in retrieving the head of the batch. We study these two algorithms:

Move to Front (MTF). MTF picks up the first element in the batch and searches for it in the list. The list cost to MTF for this search is the position at which it is found in the list. This element is then moved up to the front of the list. MTF now looks at the next batch element, searches for it in the list, and so on. MTF does not incur any batch cost.

Batch Search (BATCH). BATCH picks up the first element in the batch and searches for it in the list. So far, the cost to batch is the position where found. Now, BATCH scans the batch to look for copies of this element and deletes these copies from the batch (since these queries can be answered by the list search just completed). The batch-scanning results in a batch cost of the length of the batch (minus 1). The next element to be searched for is now taken from the head of the batch. Note that BATCH does not gain by moving the element up to the front of the list, since it will not be encountered again in the batch. Later in this paper we will want to move it to front when we consider a limited batch search.

For an algorithm f, let $T_f^L(b)$ and $T_f^B(b)$ denote the list cost and batch cost incurred in processing batch b, $|b| = m$. Let $T_f(b) = \alpha T_f^L(b) + T_f^B(b)$. Define for algorithms f and g: $\Phi_{f/g} = \frac{T_f(b)}{T_g(b)}$. We say that f is γ-*competitive* with g if

$$\limsup_{b \in U^*} \Phi_{f/g} \leq \gamma$$

First, we consider the case $\alpha = 1$. Note that the 2-competitive result in [14] cannot be directly applied to prove our result. In [14], it is assumed that the (potentially) off-line algorithm matches MTF move for move, a property that does not hold for a batch-scanning algorithm, which can eliminate duplicate queries and thus reduce the number of list searches required.

Theorem.

$$\limsup_{b \in U^*} \Phi_{\mathrm{MTF}/\mathrm{BATCH}} = 2,$$

i.e., MTF always performs within a factor of two of BATCH and there exist batches that make the competitive factor arbitrarily close to two.

We will need some notation for the proof. Without loss of generality, assume that U is the set of integers. Consider the batch b and define the following:

- Let k_i be the number of copies of element i, $1 \leq i \leq n$, in the batch. Thus if the batch is 113138 (a batch of size $m = 6$), then $k_1 = 3$ and $k_4 = 0$.
- Let d_j be the j-th distinct item scanning the batch from left to right; the head of the batch is the leftmost element in the batch. In the above example, $d_1 = 1, d_2 = 3, d_3 = 8$.
- Let a_j be the list cost for item d_j.
- Let c be the number of distinct elements in the batch.
- Let $\bar{k}_j = k_{d_j}$, i.e., the number of copies of the j-th distinct element to appear in the batch.

Proof. We first compute the cost for BATCH. For the very first item, the list cost is a_1 and the batch cost is $m - 1$. For the next item, the list cost is a_2 and the batch cost is $m - \bar{k}_1 - 1$, since the scan

would have removed $\bar{k}_1$ items and one has been removed as the first item already. Next, note that $m = \sum_{i=1}^{c} \bar{k}_i$. Substituting the batch costs above, we get a batch cost of $\sum_{i=1}^{c} \bar{k}_i - 1$ for the first item. For the second, we get $\sum_{i=2}^{c} \bar{k}_i - 1$. And so on. Now add up all the costs and we get the cost for BATCH:
$\sum_{i=1}^{c} a_i + \sum_{i=1}^{c} i\bar{k}_i \quad - c.$
Note that the total list cost for BATCH is not affected by the order of appearance of the elements in batch. To minimize the batch cost for BATCH, take $\bar{k}_1 \geq \bar{k}_2 \geq \ldots \geq \bar{k}_c$. Without loss of generality, we can assume $d_i = i$, since we are only re-labeling elements. This allows the convenient substitution $\bar{k}_i = k_i$.
To create the worst possible batch for MTF, we will show the elements must be distributed as evenly as possible in the batch. First, recall that there are at most c distinct items. Thus, through the progress of the algorithm, some list costs for MTF will be c, some list costs will be $c - 1$, and so on. Let y_i be a variable representing the number of accesses by MTF that cost i. Then the cost for MTF is $y_1 + 2y_2 + \ldots + (c-1)y_{c-1} + cy_c$. To maximize this linear cost, we will create a batch or string of accesses that first maximizes y_c, then maximizes y_{c-1}, and so on, resulting in the worst case for MTF. In order to create an access cost of c, we must cause MTF to cycle through c items in the batch — we can do this cycling at most $k_c - 1$ times. Thus, $y_c \leq ck_c$. After this, all c's will have been removed from the batch and so list costs will be at most $c - 1$. Again, to cause a list cost of $c - 1$ we must cycle through $c - 1$ items. Since there are only $k_{c-1} - k_c$ $(c-1)$-items left, $y_{c-1} \leq (c-1)(k_{c-1} - k_c)$. We repeat this argument to complete the string, as shown below.
Consider the string consisting of k_1 1's:

$$\underset{1}{1} \quad \underset{2}{1} \quad \ldots \quad \underset{k_1}{1}$$

We will create a new string by interleaving k_2 2's from the left:

$$1 \ \underset{1}{2} \ 1 \ \underset{2}{2} \ \ldots 1 \ \underset{k_2}{2} \quad 11 \ldots 1$$

Since $k_1 \geq k_2$, there may be more 1's left over at the right end. Next, interleave the k_3 3's from the left end:

$$123123 \ldots 1231212 \ldots 1211 \ldots 1$$

Repeat this until k_c c's have been inserted. Note that such interleaving has the desired property of creating the costs y_i for a singly-linked list (it would not apply to any other data structure).
We now observe MTF's performance on this string. For the first 123...c, MTF will need at least $a_1 + \ldots + a_c$. For the next c elements the cost will be c for each of them, at a cost of c^2. This will happen for the $(k_c - 1)$ clusters of $123 \ldots c$ in our string. Now, the next few clusters will be of the type $123 \ldots (c-1)$. There are $(k_{c-1} - k_c)$ such clusters. This cost for accessing these elements will be $(c-1)^2(k_{c-1} - k_c)$. Continuing in this way the time needed for MTF can be bounded as:

$$\begin{aligned} &T_{\text{MTF}}(b) \leq \textstyle\sum_{i=1}^{c} a_i + c^2(k_c - 1) \\ &+(c-1)^2(k_{c-1} - k_c) + \ldots + (k_1 - k_2) \\ &= \textstyle\sum_{i=1}^{c} a_i - c^2 + \sum_{i=1}^{c}(2i-1)k_i. \end{aligned}$$

Now, we have already computed $T_{\text{BATCH}}(b) \geq \sum_{i=1}^{c} a_i - c + \sum_{i=1}^{c} ik_i$. Therefore, as $m \to \infty$,

$$\limsup_{b \in U^*} \Phi_{\text{MTF}/\text{BATCH}} \leq 2$$

with equality when all the k_i's are equal. ∎
Note that if the list cost factor α is not equal to unity, we simply multiply the entire numerator by α and only the first term of $T_{\text{BATCH}}(b)$. Thus, we have:

Corollary.

$$\limsup_{b \in U^*} \Phi_{\text{MTF}/\text{BATCH}} = 2\alpha.$$

Figure 1: MTF(w) vs. MTF for various window sizes w ($\alpha = 2$)

Figure 2: MTF(w) vs. MTF for various window sizes w ($\alpha = 10$)

Note that we could easily improve BATCH so that elements that are once accessed are moved to the end or discarded, since they are not accessed again. However, this modification will require some overhead and we do not consider it here. Note that such a modification will not essentially affect the above results.

Our theoretical results further add to the evidence [1, 14] affirming that MTF is a simple and effective adjustment rule. Simulations generally show that MTF is always better than BATCH when $\alpha \leq 1$. However, BATCH can be effective when $\alpha >> 1$, particularly when the batch size m is large. We next consider the possibility of using limited lookahead in scanning the batch.

4 Limited Lookahead

We have thus far studied an algorithm (BATCH) that searches through the entire batch scoring off copies of the currently accessed item. We now consider the possibility of using limited lookahead – scanning only a fixed number of items in the batch starting from the head of the batch. We will refer to the limited lookahead as the *window* (into the future) and the number of items searched in the batch for each list access as the *window size*. Since the batch size may be much larger than the window size, we consider the case in which the currently accessed item is moved to the front of the list, and denote this algorithm by MTF(w), where w is the window size. Thus, loosely speaking, MTF = MTF(0) and BATCH = MTF(∞).

When comparing MTF(w) for various w with α fixed, it seems natural that some w will result in optimal performance. More precisely, if we define

Figure 3: MTF(w) vs. MTF for various window sizes w ($\alpha = 1.1$)

$D(w) = (1/m)E\left[T_{\mathrm{MTF}} - T_{\mathrm{MTF}(w)}\right]$, then we wish to find $\max_w D(w)$. Figures 1-3 plot an estimate of $D(w)$, using different values of α, for a random m-sized batch made up of independent accesses from a Zipf distribution. Here $n = 50$, $m = 100$ and $D(w)$ is plotted using an average from 10 random lists and 1000 batches per list. We plot the difference, $D(w)$, following variance reduction techniques discussed in [12] (confidence intervals are small enough that they are not shown here). Thus, MTF(w) is better than MTF when $D(w) > 0$.

Intuitively, the problem suggests that small values of w will not 'look ahead' enough whereas large values of w will incur too much lookahead overhead. The figures indicate the opposite: the optimum occurs at either $w = 0$ or $w = |b|$. Other simulation results not presented here also confirm this observation, which leads to the counter-intuitive conjecture that $D(w)$ is unimodal in w with the maximum occuring at one of the ends, $w = 0$ or $w = |b|$. We also conjecture that for every $\alpha > 1$, there is some $b'(\alpha)$ such that when $|b| > b'(\alpha)$, $w = |b|$ is optimal.

An intuition for the above phenomenon may be provided as follows. Consider an element $e \in U$ and suppose that we apply a window lookahead only when accessing e; we use $w = 0$ for the other elements. Next, assume that a fixed number of elements, say k_e elements, are found between each occurence of e in the batch. Clearly, as the window size (for e) is increased from 1 to k_e, the lookahead cost is increased while not incurring any benefit from looking ahead. Thus, for $1 \leq w \leq k_e$, MTF(w) is worse than MTF, independent of the value of α. Yet, we can take α large enough so that BATCH is better than MTF. In this case, as w is increased (assuming continuity), the costs of MTF(w) must eventually start decreasing, finally reaching the cost of BATCH (lower than that of MTF). Of course, we have only provided an intuition using the above contrivance. In our original model, the window lookahead is applied to all elements and, further, the number of elements between successive occurences of any element e is random.

5 Summary

In this paper we studied self-adjusting linked lists when the entire string of accesses to the list is made available a priori. We argued that this string, or *batch* as we refer to it, must itself be stored in a data structure comparable to the linked list and thus manipulating the string in order to look ahead incurs a cost roughly equal to that of manipulating the linked list. If the costs are represented by a ratio α then we showed that the well-known Move to Front algorithm is 2α-competitive with an algorithm that always scans the batch. When considering limited lookahead, we have simulation evidence of the counter-intuitive observation that the optimal strategy is to either search the entire batch or not search at all. Some open problems are suggested by our work. First, it would be interesting to have a simple probabilistic model explain the above counter-intuitive result. Secondly, a formal analysis of heuristics that dynamically adjust the window size (lookahead amount) would be desirable.

Acknowledgement. We would like to thank Prof. Weizhen Mao, College of William & Mary, for some valuable comments and suggestions.

References

[1] J.L.Bentley and C.C.McGeoch. Amortized Analyses of Self-Organizing Sequential Search Heuristics. *CACM*, Vol. 28, No. 4, April 1985, pp. 404-411.

[2] J.R.Bitner. Heuristics that Dynamically Organize Data Structures. *SIAM J. Computing*, Vol. 8, No. 1, February 1979, pp. 82-109.

[3] P.J.Burville and J.F.C.Kingman. On a Model for Storage and Search. *J. Appl. Prob.*, Vol. 10, 1973, pp. 697-701.

[4] H.T.Ch'ng, B.Srinivasan and B.C.Ooi. Study of Self-Organizing Heuristics for Skewed Access Patterns. *Info. Proc. Lett.*, Vol. 30, 1989, pp. 237-244.

[5] F.R.K.Chung, D.J.Hajela and P.D.Seymour. Self-organizing Sequential Search and Hilbert's Inequalities. *Proc. ACM STOC*, 1985, pp. 217-223.

[6] G.H.Gonnet, I.Munro and H.Suwanda. Exegesis of Self-Organizing Linear Search. *SIAM J. Computing*, Vol. 10, No. 3, August 1981, pp. 613-637.

[7] W.J.Hendricks. An Extension of a Theorem Concerning an Interesting Markov Chain. *J. Appl. Prob.*, Vol. 10, 1973, pp. 886-890.

[8] J.H.Hester and D.S.Hirschberg. Self-Organizing Linear Search. *Computing Surveys*, Vol. 17, No. 3, September 1985, pp. 295-311.

[9] M.Hofri and H.Shachnai. On the Limited Utility of Auxiliary Information in the List Update Problem. *Stochastic Models*, Vol. 8, No. 4, 1992, pp 637-650.

[10] M.S.Manasse, L.A.McGeoch and D.D.Sleator. Competitive Algorithms for On-line Problems. *J.Algorithms*, Vol. 11, 1990, pp. 208-230.

[11] J.McCabe. On a Serial File with Relocatable Records. *Operations Research*, Vol. 12, 1965, 609-618.

[12] C.McGeoch. Analyzing Algorithms by Simulation: Variance Reduction Techniques and Simulation Speedups. *ACM Computing Surveys*, Vol. 24, No. 2, 1992, pp. 195-212.

[13] R.Rivest. On Self-Organizing Sequential Search Heuristics. *CACM*, Vol. 19, 1976, pp. 63-67.

[14] D.D.Sleator and R.E.Tarjan. Amortized Efficiency of List Update and Paging Rules. *CACM*, Vol. 28, No. 2, February 1985, pp. 202-208.

[15] R.Tarjan. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[16] R.S.Valiveti and B.J.Oommen. The Move-to-Front List Organizing Heuristic for Non-Stationary Query Distributions. *Proc. Int. Symp. Comp. Info. Sci.*, 1991, Antalya, Turkey, pp. 105-114.

[17] R.S.Valiveti, B.J.Oommen and J.Zgierski. Adaptive List Reorganization for a System Processing Set Queries. *Proc. Conf. Fundamentals of Computation Theory*, September 1991, Berlin, pp. 405-414.