Amortized Efficiency of List Update Rules

Daniel Dominic Sleator
Robert Endre Tarjan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

In this paper we study the amortized complexity of two well-known algorithms used in system software. These are the "move-to-front" rule for maintaining an unsorted linear list representing a set, and the "least-recently-used" replacement rule for reducing page faults in a two-level paged memory. These algorithms have been subjected to much analysis, most of it average case. By studying the amortized complexity of these algorithms we are able to gain additional insight into their behavior.

By amortization we mean averaging the running time of an algorithm over a worst-case sequence of operations. An amortized bound on the running time of an algorithm tells much more than an average-case bound of similar magnitude. This is because an amortized bound constrains the cost of a worst-case sequence of operations, whereas an average-case bound only tells what will probably happen, and only applies under probabilistic assumptions. Furthermore in many situations an amortized bound is more useful than a worst-case per operation bound, because a worst-case bound cannot take into account correlated behavior among different operations. In these situations a worst-case bound per operation is overly pessimistic.

In section 2 of this paper we analyze the amortized efficiency of the move-to-front list update rule, under the assumption that accessing the i^{th} element from the front of the list takes $\Theta(i)$ time. We show that this algorithm has a running time within a factor of two of that of the optimum off-line list update algorithm. This means that no algorithm, on-line or not, can beat move-to-front by more than a constant factor, on any sequence of operations. We also show that the "move-half-way-to-front" rule satisfies a similar bound but with a constant factor of four instead of two. Other common heuristics, such as the transpose and frequency count rules, do not share this approximate optimality.

In section 3 we state our results about paging. There is a close analogy between the "least-recently-used" (LRU) rule for paging and the move-to-front rule for list updating. In fact the running time of LRU can be analyzed by considering the move-to-front rule for updating a list with the following cost measure: the cost to access the i^{th} item in the list is one if $i>n$ and zero otherwise, where n is the size of the paged memory. Although LRU is not within a constant factor of optimum, we are able to show that its amortized cost differs from the optimum by a factor that depends on the size of fast memory, and that no on-line algorithm has better amortized performance. Section 4 contains some concluding remarks.

2. Self-Organizing Lists

The problem we shall study in this paper is often called the *dictionary problem:* maintain a set of items under an intermixed sequence of the following three kinds of operations:

access(i): Locate item i in the set.

insert(i): Insert item i into the set.

delete(i): Delete item i from the set.

In discussing this problem, we shall use n to denote the maximum number of items ever in the set at one time and m to denote the total number of operations. We shall generally assume that the initial set is empty.

A simple way to solve the dictionary problem is to represent the set by an unsorted list. To access an item, we scan the list from the front until locating the item. To insert an item, we scan the entire list to verify that the item is not already present and then insert it at the rear of the list. To delete an item, we scan the list from the front to find the item and then delete it. In addition to performing access, insert, and delete operations, we may occasionally want to rearrange the list by exchanging pairs of consecutive items. This can speed up later access operations.

We shall consider only algorithms that solve the dictionary problem in the manner described above. In particular, as a list rearrangement operation we allow only the exchange of *adjacant* items. We divide the cost of an operation into two parts, the number of comparisons and

the number of exchanges. We separate these two costs because the relative cost of an exchange versus a comparison depends on the implementation of the list. Accessing or deleting the i^{th} item in the list requires i comparisons. Inserting a new item takes $i+1$ comparisons, where i is the size of the list before the insertion. (We charge a comparison for locating the end of the list.)

Our goal is to find a simple rule for updating the list (by performing exchanges) that will make the total cost of a sequence of operations as small as possible. Three rules have been extensively studied, under the rubric of *self-organizing linear lists:*

> *Move-to-front* (MF). After accessing or inserting an item, move it to the front of the list, without changing the relative order of the other items.
>
> *Transpose* (T). After accessing or inserting an item, exchange it with the immediately preceding item.
>
> *Frequency count* (FC). Maintain a frequency count for each item, initially zero. Increase the count of an item by one whenever it is inserted or accessed; reduce its count to zero when it is deleted. Maintain the list so that the items are in non-increasing order by frequency count.

Bentley and McGoech's paper on self-organizing lists [3] contains a summary of previous results. These deal mainly with the case of a fixed set of n items on which only accesses are permitted and exchanges are not counted. For our purposes the most interesting results are the following. Suppose the accesses are independent random variables and that the probability of accessing item i is p_i. For any algorithm A let $E_A(p)$ be the asymptotic expected cost of an access, where $p=(p_1,p_2,..,p_n)$. In this situation, an optimum algorithm, which we call *decreasing probability* (DP), is to use a fixed list with the items arranged in non-increasing order by probability. The strong law of large numbers implies that $E_{FC}(p)/E_{DP}(p) = 1$ for any probability distribution p [8]. It has long been known that $E_{MF}(p)/E_{DP}(p) \le 2$ [3,7]. Rivest [8] showed that $E_T(p) \le E_{MF}(p)$, with the inequality strict unless $n = 2$ or $p_i = 1/n$ for all i. He further conjectured that transpose minimizes the expected access time for any p, but Anderson, Nash and Weber [1] found a counterexample.

In spite of this theoretical support for transpose, move-to-front performs much better in practice. One reason for this, discovered by Bitner [4], is that move-to-front converges much faster to its asymptotic behavior if the initial list is random. A more compelling reason was discovered by Bentley and McGoech [3], who studied the amortized complexity of list update rules. Again let us consider the case of a fixed list of n items on which only accesses are permitted, but let s be any sequence of access operations. For any algorithm A, let $C_A(s)$ be the total number of comparisons done. Bentley and McGoech compared move-to-front, transpose and frequency count to the optimum static algorithm, called *decreasing frequency* (DF), which uses a fixed list with the items arranged in non-increasing order by access frequency. Among algorithms that do no rearranging of the list, decreasing frequency minimizes the total access cost. Bentley and McGoech proved that $C_{MF}(s) \le 2C_{DF}(s)$ if MF's initial list contains the items in order by first access. Frequency count but not transpose shares this property. A counterexample for transpose is an access sequence consisting of a single access to each of the n items followed by repeated accesses to the last two items, alternating between them. On this sequence transpose costs $mn-O(n^2)$, whereas decreasing frequency costs $1.5m+O(n^2)$.

Bentley and McGoech also tested the various update rules empirically. Their tests show that transpose is inferior to frequency count but move-to-front is competitive with frequency count and sometimes better. This suggests that some real sequences have a high locality of reference, which move-to-front but not frequency count exploits. Our first theorem, which generalizes Bentley and McGoech's theoretical results, helps to explain this phenomenon.

For any algorithm A and any sequence of operations s, let $C_A(s)$ be the total number of comparisons, and let $X_A(s)$ be the total number of exchanges.

Theorem 1. For any algorithm A and any sequence of operations s starting with the empty set,

$$C_{MF}(s) \le 2C_A(s)+X_A(s).$$

Proof. We use the concept of a potential function that measures the difference between MF's list and A's list. Consider running algorithms A and MF in parallel on s. A potential function maps a configuration of A's and MF's lists onto a real number Φ. If we do an operation that takes time t for the MF algorithm and changes the configuration fron one with potential Φ to one with potential Φ', we define the *amortized time* of the operation to be $t+\Phi'-\Phi$. That is, the amortized time of an operation is its running time plus the increase it causes in the potential. If we perform a sequence of operations such that the i^{th} one takes actual time t_i and has amortized time a_i, then when we sum the amortized times the potential changes telescope to give the following relationship:

$$\sum_i t_i = \Phi-\Phi'+\sum_i a_i$$

where Φ is the initial potential and Φ' is the final potential. Thus we can estimate the total running time by choosing a potential function and bounding Φ, Φ', and a_i for each i. (Note that for the purposes of this proof "time" is a measure of the number of comparisons done.)

To obtain the theorem we let the potential function be the number of inversions in MF's list with respect to A's list. This is the number of pairs of items that are in different relative order in MF's list than they are in A's list. With this potential function we shall show that the amortized time for MF to access item i is at most $2i-1$, the amortized time for MF to insert an item into a list of size i is at most $2(i+1)-1$, and the amortized time for MF to delete item i is at most i, where we identify an item by its

position in A's list. Furthermore the amortized time charged to MF when A does an exchange is at most one.

The initial configuration has a potential of zero since the initial lists are empty, and the final configuration has a non-negative potential. Thus the actual cost of a sequence of operations to MF is bounded by the sum of the amortized times of the operations. The sum of the amortized times is in turn bounded by the right hand side of the inequality we wish to prove. (The bounds stated in the previous paragraph imply that for any operation the amortized time for MF to do the operation is at most twice the actual time used by A to do the operation.)

It remains for us to bound the amortized times of the operations. Consider an access by A to an item i. Let k be the position of i in MF's list and let x be the number of items that precede i in MF's list but follow i in A's list. (See Figure 1.) Then the number of items preceding i in both lists is $k-1-x$. Moving i to the front of MF's list creates $k-1-x$ inversions and destroys x other inversions. The amortized time for the operation (the time for MF to do the operation plus the increase in the number of inversions) is therefore $k+(k-1-x)-x = 2(k-x)-1$. But $k-x \leq i$ follows from $k \leq i+x$ which follows from the fact that all the items preceeding i in MF's list either come before or after i in A's list. Thus the amortized time for an access is at most $2i-1$.

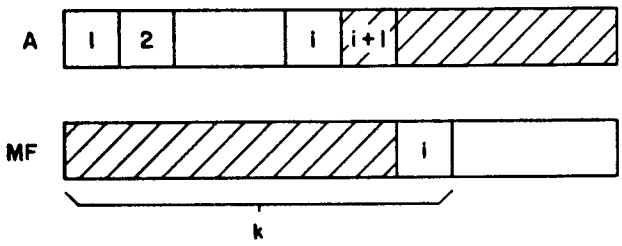

Figure 1. Arrangement of A's and MF's lists in the proof of Theorem 1. The number of items common to both shaded regions is x.

The arguments for insertion and deletion are virtually the same as that for access. In the case of deletion no new inversions are created, and the amortized time is $k-x \leq i$.

An exchange by A takes zero actual time for MF, so the amortized time is just the increase in the number of inversions caused by the exchange. This increase is at most 1. Q.E.D.

To see that the factor of two in this theorem is tight, let A be the algorithm that does no exchanges, and let s be a sequence that always searches for the last item in MF's list. In this case $C_{MF}(s)=mn$ and $C_A(s)=m(\frac{1+n}{2})$, since all the items are accessed equally often.

For the sake of simplicity we have not included here a proof of the following stronger result:

$$C_{MF}(s) \leq 2C_A(s)+X1_A(s)-X2_A(s)-m.$$

Here the exchanges done by A are divided into two classes. The number of exchanges that are done immediately after an access or an insertion of item i which move i closer to the front of the list is $X2_A(s)$, The number of other exchanges is $X1_A(s)$. (Recall that m is the total number of operations done.)

Theorem 1 generalizes to the situation in which the initial set is non-empty and MF and A begin with different lists (containing the same items in different orders). In this case the bound is

$$C_{MF}(s) \leq 2C_A(s)+X1_A(s)-X2_A(s)+I-m.$$

Where I is the number of inversions separating A's initial list and MF's initial list, which is at most $\binom{n}{2}$.

We can use Theorem 1 to bound the cost of MF when both comparisons and exchanges are counted. If we let $T_A(s)$ be the number of comparisons plus the number of exchanges, then Theorem 1 implies that

$$T_{MF}(s) \leq 4T_A(s).$$

This follows from $T_A(s) = C_A(s)+X_A(s)$ and $X_{MF}(s) \leq C_{MF}(s)$.

The proof of Theorem 1 applies to any update rule in which the accessed item is moved some fraction of the way to the front, as the following theorem shows.

Theorem 2. If MF(d) ($d \geq 1$) is any rule that moves an accessed or inserted item in position k at least $\frac{k}{d}-1$ units closer to the front of the list then

$$C_{MF(d)}(s) \leq 2dC_A(s)+dX_A(s)-dm.$$

Proof. The proof follows the same outline as that of Theorem 1. The potential function we use is d times the number of inversions separating A's and MF(d)'s lists. We shall show that the amortized time for MF(d) to access an item i (in position i in A's list) is at most $2di-d$, the amortized time for MF(d) to insert an item into a list of size i is at most $2d(i+1)-d$, and the amortized time for MF(d) to delete item i is at most i. Furthermore the amortized time charged to MF(d) when A does an exchange is at most d. These bounds are used as in the proof of Theorem 1 to give the result.

Consider an access to item i. Let k be the position of i in MF(d)'s list, and p be the distance that MF(d) moves item i (the number of items past which i is pushed). Let x be the number of these items that occur after i in A's list. (See Figure 2.) The decrease in the number of inversions caused by this move is x, while the increase is $p-x$. Thus the potential function increases by $d(p-2x)$. The amortized time is thus $k+dp-2dx$. We want to show that this is less than or equal to $2di-d$. The inequalities we have to work with are

$$\frac{k}{d}-1 \leq p \leq x+i-1.$$

The first inequality follows from the definition of MF(d)'s rule. The second follows from the fact that each of the p items that i moves past is either from a set of $i-1$ items

less than i, or from a set of x items greater than i. If we multiply the inequalities through by d use transitivity we get

$$k \leq dx+di.$$

Adding the second inequality multiplied by d gives

$$k+dp \leq 2dx+2di-d.$$

Subtracting $2dx$ from both sides yields

$$k+dp-2dx \leq 2di-d,$$

which is the desired result.

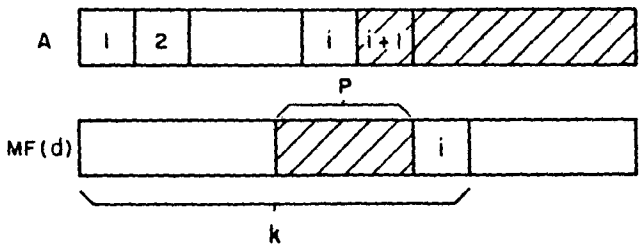

Figure 2. Arrangement of A's and MF(d)'s list in the proof of Theorem 2. The number of items common to both shaded regions is x.

A similar argument applies to insertion. In the case of deletion the amortized time is at most $k-dx$ (where x is as shown in Figure 1). This is at most $k-x$, which in turn is bounded by i as shown in the proof of Theorem 1. Since i and d are at least 1, $i \leq 2i-1 \leq 2di-d$. Finally, an exchange done by algorithm A increases the potential by at most d. (Note that the $-d$ terms in the amortized times accumulate to form the term $-dm$ in the theorem.) Q.E.D.

No analagous result holds for transpose or for frequency count. In both cases there are counterexamples that show that they can be worse than MF by an arbitrarily large factor.

Theorem 1 is a very strong result, stating that on *any* sequence of operations, move-to-front is to within a constant factor as efficient as *any* algorithm, including algorithms that base their behavior on advance knowledge of the entire sequence of operations. (If the operations must be performed on-line, such off-line algorithms are unusable.) This is the only non-trivial example we know of in which a particular algorithm is as good as any other algorithm on any input.

More quantitatively, Theorem 1 provides us with a way to measure the inherent complexity of a sequence. Suppose we begin with the empty set and perform a sequence of insertions and accesses. We define the complexity of an access or insert operation on item i to be to be one plus the number of items accessed or inserted since the last operation on i. The complexity of the sequence is the sum of the complexities of its individual operations. With this definition the complexity of a sequence is exactly the cost of move-to-front on the sequence and is within a factor of two of the cost of an optimum algorithm.

These theorems can be generalized further to encompass the situation in which the cost of an access is a convex function of the position. These generalizations will be in our full paper.

3. Paging

Consider a two-level memory divided into *pages* of fixed uniform size. Let n be the number of pages of fast memory. Each operation is an *access* that specifies a page of information. If the page is in fast memory, the access costs nothing. If the page is in slow memory we must swap it for a page in fast memory, at the cost of one *page fault*. Our goal is to minimize the number of page faults for a given sequence of accesses.

Such a two-level memory corresponds to a self-organizing list with the following access cost: accessing i costs 0 if $i \leq n$ and 1 otherwise. The only difference between the paging problem and the corresponding list update problem is that a page in slow memory *must* be moved to fast memory when accessed, whereas the corresponding list reordering is optional. We shall use standard paging terminology and require that each accessed page be moved to fast memory.

As with list updating, most previous work on paging [2,5,6,9] is average case analysis. Among paging rules that have been studied are the following:

Least-recently-used (LRU) When a page fault occurs, replace the page whose most recent access was earliest.

First-in, first-out (FIFO) Replace the page that has been in fast memory the longest.

Last-in, first-out (LIFO) Replace the page most recently moved to fast memory.

Least-frequently-used (LFU) Replace the page that has been accessed the least.

Longest-forward-distance (MI_1^1) Replace the page whose next access is the latest.

Least-recently-used paging is equivalent to move-to-front list updating, and least-frequently-used paging corresponds to frequency count. All the paging rules except longest-forward-distance are on-line algorithms; that is, they require no knowledge of future accesses. Longest-forward-distance exactly minimizes the number of page faults [2], which is why it is known as the MIN algorithm.

We shall compare various algorithms to the MIN algorithm. In making such a comparison, it is revealing to let the two algorithms have different fast memory sizes. If

A is any algorithm and s is any sequence of m accesses, we denote by n_A the number of pages in A's fast memory and by $F_A(s)$ the number of page faults made by A on s. When comparing A and MIN, we shall assume that $n_A \geq n_{MIN}$.

Theorem 3. Let A be any on-line algorithm. Then there are arbitrarily long sequences s such that

$$F_A(s) \geq \left(\frac{n_A}{n_A - n_{MIN} + 1}\right) F_{MIN}(s)$$

Theorem 4. For any sequence s,

$$F_{LRU}(s) \geq \left(\frac{n_{LRU}}{n_{LRU} - n_{MIN} + 1}\right) F_{MIN}(s) + n_{MIN}.$$

Theorem 4 applies to FIFO as well as LRU. The proofs of these two theorems will appear in our full paper. These results can be interpreted either positively or negatively. On the one hand, any on-line paging algorithm makes a number of faults in the worst case that exceeds the minimum by a factor equal to the size of fast memory. On the other hand, even in the worst case both LRU and FIFO paging come within a constant factor of the number of page faults made by the optimum algorithm with a constant factor smaller fast memory. More precisely, for any constant factor $c > 1$, LRU and FIFO with fast memory size n make at most c times as many faults as the optimum algorithm with memory size $(1 - \frac{1}{c})n$. A similar result for LRU was proved by Franaszek and Wagner [6], but only on the average.

4. Remarks

We have studied the amortized complexity of the move-to-front rule for list updating, showing that on any dequence of operations it has a total cost within a constant factor of minimum among all possible updating rules, including off-line ones. The constant factor is two if we do not count the updating cost incurred by move-to-front and four if we do. This result is much stronger than previous average-case results on list update heuristics. Neither transpose nor frequency count shares this approximate optimality. Thus even if one is willing to incur the time and space overhead needed to maintain frequency counts, it may not be a good idea.

Our results lend theoretical support to Bentley and McGoech's experiments showing that move-to-front is generally the best rule in practice. As Bentley and McGoech note, this contradicts the asymptotic average-case results, which favor transpose over move-to-front. Our tentative conclusion is that amortized complexity provides not only a more robust but a more realistic measure for list update rules than asymptotic average-case complexity.

References

[1] E. J. Anderson, P. Nash, and R. R. Weber, "A conterexample to a conjecture on optimal list ordering," *J. Appl. Prob.*, to appear.

[2] L. A. Belady, "A study of replacement algorithms for vitrual storage computers," *IBM Sys. J.*, **5** (1966), 78-101.

[3] J. L. Bentley and C. McGoech, "Worst-case analysis of self-organizing sequential search heuristics," *Proc. 20th Allerton Conference on Communication, Control, and Computing*, 1982.

[4] J. R. Bitner, "Heuristics that dynamically organize data structures," *SIAM J. Computing*, **8** (1979), 82-110.

[5] E. J. Coffman and P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[6] P. A. Franaszek and T. J. Wagner, "Some distribution-free aspects of paging performance," *J. ACM*, **21** (1974), 31-39.

[7] D. E. Knuth, *The Art of Computer Programming, Volume 1: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[8] R. Rivest, "On self-organizing sequential search heuristics," *Comm ACM*, **19** (1976), 63-67.

[9] J. R. Spirn, *Program Behavior: Models and Measurements*, Elsevier, New York, 1977.