Yale University
Department of Computer Science

Optimum Off-line Algorithms for The List Update Problem

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

YALEU/DCS/TR-805
August 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. Research partially supported by NSF grant CCR-9009753

Optimum Off-line Algorithms for The List Update Problem

Nick Reingold[*] *Jeffery Westbrook*[†]

Abstract

We investigate optimum off-line algorithms for the list update problem, a well-known problem of implementing a dictionary of items as a linear list. We give several characterizations of optimum algorithms; our results lead to an implementation of an optimum algorithm which runs in time $O(2^n n! m)$, where n is the length of the list and m is the number or requests. The previous best algorithm runs in time $O((n!)^2 m)$ [6]. Off-line algorithms for list update are useful in the study of competitive on-line algorithms.

1 Introduction

Storing a dictionary of items is a common problem in many programming situations. Since many operations, such as *accesses*, *insertions*, and *deletions* are very simple to perform on linear lists, it is natural to consider dictionaries stored using this data structure. An access can easily be done by examining each item in turn, starting at the front of the list, until either finding the item desired or reaching the end of the list and reporting the item not present. During the course of performing a sequence of operations on a list, it may be advantageous to rearrange the items so that later operations will be cheaper to perform. For example, by moving a commonly requested item closer to the front the overall running time can be reduced. On the other hand, performing a rearrangement may itself be costly. Finding an algorithm to rearrange the items most effectively is an interesting and difficult problem. This problem, called the *list update* problem, has been investigated by many researchers [1, 3, 5, 7, 9, 10].

A list update algorithm takes a request sequence and determines rearrangements to be done between processing accesses, deletions, or insertions. We distinguish among

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

[†]Department of Computer Science, Yale University, New Haven, CT 06520-2158. Research partially supported by NSF grant CCR-9009753

list update algorithms based on how much knowledge of the future they are given. An *on-line* algorithm knows only the sequence of requests it has processed up to the current time, and has no knowledge of the future requests. An *off-line* algorithm knows the entire sequence of requests. One simple off-line algorithm is to initially order the items by decreasing frequency of access, and after that make no rearrangements. This is the optimum *static* rule. That is, among rules which do no rearrangements during the processing of the sequence, this rule has the lowest total cost for any sequence of accesses. This rule can be used to build an on-line algorithm if the expected access frequency is known for each item. Of course, it may not be easy or even possible to determine the frequency counts.

Most real-world applications are on-line, and many on-line rules have been studied. Typically, their performance is analyzed not in absolute terms, but in comparison to that of the optimum static rule ([1, 3, 5, 7, 9]). One of the earliest rules studied is the Move-To-Front rule (MTF): after an item is accessed, move it to the front of the list. Sleator and Tarjan [10] compared MTF to *any* algorithm. They showed that for any sequence of insertions, deletions, and accesses, the cost to MTF is no more than twice the cost of any list update algorithm for servicing that request sequence. This result initiated competitive analysis, in which an on-line algorithm's performance is compared to that of the optimum off-line algorithm on each sequence of requests. In another paper [8] we investigate the problem of randomized on-line algorithms for the list update problem.

In this report we investigate the problem of finding the optimum off-line algorithm for any sequence of requests. That is, we are trying to find an algorithm that, given a request sequence, computes a set of rearrangements prior to each access that minimizes the overall total cost of processing that request sequence. The cost of processing a sequence is the sum of the costs of accesses, insertions, deletions, and rearrangements. The running time of the algorithm that computes the optimum set of rearrangements is a separate issue. Finding a good optimum off-line algorithm is an interesting problem in its own right, but a perhaps more important goal of this research is to help in the design of on-line algorithms. The better the optimum off-line algorithm is understood the better we can make on-line algorithms simulate its behavior. In particular, we improve the theoretical understanding of why the move-to-front heuristic for on-line algorithms leads to good competitive ratios while other heuristics do not. Furthermore, the results of this research have been used in [8] to find lower bounds on the competitiveness of randomized on-line algorithms for list update. This research is also motivated by practical uses, since to perform empirical studies of the competitiveness of on-line algorithms for list-update it is necessary to have an optimum or even near-optimum off-line algorithm to compare against.

For the related k-server problem, Chrobak, *et al* [4] have given an $O(km^2)$ algorithm to compute the best strategy for servicing m requests. Before our results, the best algorithm to compute the optimum cost for the list update problem was a version of an algorithm of Manasse, McGeoch, and Sleator [6] which runs in time

$\Theta(m(n!)^2)$, and space $\Theta(n!)$, where m is the number of requests and n is the number of items in the list. In this paper we give an improved algorithm for list update running in worst-case time $O(m2^n(n-1)!)$ and space $\Theta(n!)$. Thus we still leave open the problem of finding an algorithm that is polynomial in n and m.

This paper is organized as follows. Section 2 contains a formal description of the list update model as defined by Sleator and Tarjan [10]. We show one simplification that can be made to the model. We also give a counter-example to a theorem of Sleator and Tarjan from [10]. Section 3 contains various theorems about the characteristics of optimum algorithms. The most important is the Subset Transfer Theorem, which implies that in searching for an optimum way to service a request sequence, we may assume that the rearrangements made are of a particular sort: a subset of the items ahead of the requested item are moved to just behind the requested item. This allows a significant speedup in searching for the optimum algorithm. We discuss this and other implementation issues in Section 4. Section 5 contains some remarks and open questions.

2 The List Update Model

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. 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.

In the *standard model* defined by Sleator and Tarjan [10] the costs are as follows:

1. An access or deletion of the i^{th} item in the list costs i. An insertion costs $n+1$, where n is the length of the list prior to the insertion.

2. Immediately following an access or insertion, the accessed or inserted item can be moved any distance forward in the list. These exchanges cost nothing and are called *free* exchanges.

3. Any other exchange costs 1. These exchanges are called *paid* exchanges.

We can think of an algorithm as servicing each request by performing some number of paid exchanges, receiving and satisfying the request, and then doing any free exchanges. For a list update algorithm, A, we define the *cost* of servicing a sequence of requests σ to be the sum of all costs due to paid exchanges, accesses, insertions, and deletions in the sequence. Let $A(\sigma)$ denote 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. For any request sequence, σ, there is a minimum cost which can be incurred in servicing σ, which we denote by $\mathrm{OPT}(\sigma)$. If a list update algorithm, A, is such that $A(\sigma) = \mathrm{OPT}(\sigma)$ for all σ, we say that A is an *optimum* list update algorithm. Notice that in general there are several ways to service a request sequence and achieve the optimum cost.

We also consider a generalization of the standard model called the *paid exchange* model, in which the access cost is the same as the standard model but in which there are no free exchanges and the cost of rearrangement is scaled up by some arbitrarily large value d. This is a very natural extension, because there is no a priori reason to assume that the execution time of the program that swaps a pair of adjacent elements is the same as that of the program that does one iteration of the search loop. (This might model a linear list on a tape drive). We denote by P^d the paid exchange model in which each paid exchange has cost d. Accesses, insertions, and deletions have the same cost as in the standard model. This version of the list update problem is similar to the replication/migration problems studied by Black and Sleator [2].

Most standard model on-line algorithms use only free exchanges, and it is tempting to conclude that with free exchanges available, paid exchanges should never be needed. Sleator and Tarjan claim (Theorem 3 of [10]) that in the standard model, there is a minimum cost off-line algorithm that uses only free exchanges. Unfortunately, their theorem is wrong. In fact, paid exchanges are sometimes necessary, as the following counterexample shows: If the list is initially $1, 2, 3$ and the request sequence is $\langle 3, 2, 2, 3 \rangle$, it is easy to verify that any algorithm which makes only free exchanges must incur a cost of at least 9. On the other hand, two paid exchanges can make the list look like $2, 3, 1$, and then the accesses can be performed at an additional cost of only 6, for a total cost of 8.

On the other hand, we can show that *free* exchanges are not necessary in the standard model.

Proposition 2.1 *Let R be a request sequence, and A be any algorithm for servicing R in the standard model. Then there is an algorithm, $\hat{A}$, for serving R which makes no free exchanges and which costs exactly the same as A on R.*

Proof We modify A to eliminate free exchanges as follows. Suppose that on some request A makes free exchanges to move item x forward ℓ places. $\hat{A}$ will make paid exchanges to move item x forward ℓ places *just before the request*, and then make no free exchanges after the request. $\hat{A}$ spends ℓ in extra paid exchanges to move x forward, but saves ℓ on the access. ∎

This proposition implies that the standard model and P^1 are equivalent for off-line algorithms in the sense that given a request sequence σ, the minimum off-line cost to service σ is the same in both models. In general, if a list update algorithm is charged $f(i)$ to access the i^{th} item, then the proposition is true if the cost of exchanging the

i^{th} and the $(i+1)^{\text{st}}$ items is $f(i+1) - f(i)$. Notice that Proposition 2.1 is false in any P^d model with $d > 1$.

3 Characteristics of Off-line Algorithms

In this section we present several theorems that characterize the behavior of off-line algorithms for list update. These theorems will be used in designing an optimum off-line algorithm. If we are exhaustively searching for what moves to make, applications of these theorem can significantly reduce the size of the search space. We will restrict our attention to off-line algorithms that do not use free exchanges, since by Proposition 2.1, in the standard model, any algorithm that uses free exchanges can be converted at no cost into one that uses only paid exchanges. Furthermore, we will for the moment assume that request sequences contain only accesses. In Section 4 we argue that an optimum strategy for a request sequence containing insertions and deletions can be determined by examining a similar request sequence consisting only of accesses.

Theorems 3.1, 3.3, and 3.5 hold in the standard model and the P^1 model. Theorem 3.7 holds in the standard model and any P^d model (paid exchanges only, with cost d per exchange).

3.1 Long Blocks

Intuitively, if some item is requested many times consecutively, it should be moved to the front of this list. In this section we prove this intuition, and give a useful generalization.

Theorem 3.1 (Long Block Theorem) *Let request sequence $R = \sigma_1 x^k \sigma_2$, where x^k denotes a block of k consecutive accesses to some item x, $k \geq 2$, and σ_1 and σ_2 are any request sequences. Let A be an off-line algorithm that services R at total cost c. Then there is an algorithm $\hat{A}$, that services R with total cost at most c and in which x is at the front of the list for each of the k consecutive accesses to x .*

Proof We may assume that A makes no free exchanges. Algorithm $\hat{A}$ mimics A at all times, except that x is moved to the front just before the first of the k accesses, and immediately after the last of the k accesses x is moved back to the place it occupies in A's list. Algorithms A and $\hat{A}$ will have identical lists just after processing σ_1 and before starting to process σ_2. Roughly speaking, the first access to x will pay for moving x forward, and the last access will pay for moveing x back.

Let p_i denote the location of x in A's list at the time of the i^{th} access in the k-block; p_0 is the position of x immediately after the last request in σ_1 is serviced and p_k is x's position immediately after the k^{th} access. Consider the sequence of moves made by A during the accesses to x; it consists of k subsequences of exchanges (which may

involve x) occurring before and between the k accesses to x. Suppose that during the i^{th} such subsequence there are β_i exchanges that involve x, and that x moves from position p_{i-1} to position p_i. Notice that $|p_{i-1} - p_i| \leq \beta_i$.

We construct $\hat{A}$'s moves by modifying A's sequence of exchanges as follows:

- Ignore any exchanges made by A that involve item x, but duplicate every other exchange A makes, at exactly the same time.
- Immediately after servicing σ_1, perform exchanges which move x from position, p_0, to the front.
- Immediately after the k^{th} access to x, perform exchanges which move x from the front to position p_k.

It is important to show that $\hat{A}$ can duplicate the exchanges made by A, *i.e.*, that moving x to the front does not interfere with other exchanges A makes. A simple induction shows, however, that at all times any pair of items, excluding x, that are adjacent A's list are also adjacent in $\hat{A}$'s list. Note that after all these exchanges are made A and $\hat{A}$ will have identical lists.

On the i^{th} sequence of paid exchanges, for $1 \leq i \leq k$, A pays β_i more than $\hat{A}$. On the i^{th} access of x, for $1 \leq i \leq k$, A pays $p_i - 1$ more than $\hat{A}$. Finally, $\hat{A}$ pays $p_0 - 1$ to move x to the front at the start, and $p_k - 1$ to move x to position p_k at the end. Thus,

$$\begin{aligned} A(x^k) - \hat{A}(x^k) &= \left(\sum_{i=1}^{k} \beta_i\right) + \left(\sum_{i=1}^{k} (p_i - 1)\right) - (p_0 - 1) - (p_k - 1) \\ &= \left(\sum_{i=2}^{k} \beta_i\right) + \left(\sum_{i=2}^{k-1} (p_i - 1)\right) + (\beta_1 + p_1 - p_0) + (p_k - p_k) \\ &\geq \left(\sum_{i=2}^{k} \beta_i\right) + \left(\sum_{i=2}^{k-1} (p_i - 1)\right) + (0) + (0) \\ &\geq 0 \end{aligned}$$

∎

Notice that if any $\beta_i > 0$ for $2 \leq i \leq k$, or if any $p_i > 1$ for $2 \leq i \leq k-1$, then $\hat{A}$ does strictly better than A. This implies the following:

Corollary 3.2 *If an item x is requested three or more times consecutively then an optimum algorithm must move it to the front before the second access.*

The following theorem generalizes the Long Block Theorem.

Theorem 3.3 *Let request sequence $R = \sigma_1 \tau \sigma_2$, where τ is an access sequence such that every item accessed in τ is accessed at least twice, and σ_1 and σ_2 are any request*

sequences. Let A be an off-line algorithm that services R at total cost c. Then there is an algorithm $\hat{A}$, that services R with total cost at most c such that after processing σ_1 and before processing τ, $\hat{A}$ moves all items accessed in τ in an order-preserving manner to the front of the list.

Proof Let $X = \{x_1, x_2, \ldots, x_r\}$ be the set of items which occur in τ. Let the number of accesses to x_i in τ be k_i. We will assume that A makes no free exchanges. The idea is similar to the idea of Theorem 3.1: move all the items mentioned in τ to the front, mimic A's moves, then restore the list to match A's. Once again, each item pays for its trips with savings on the first and last accesses to it.

For each x_i we split τ into *phases*. For $1 \leq j \leq k_i + 1$, the j^{th} phase for x_i starts just after the $(j-1)^{\text{st}}$ access to x_i (or at the start of τ if $j = 1$) and ends just before the j^{th} access to x_i (or at the end of τ if $j = k_i + 1$). Let $p(i, 0)$ be the initial position of x_i in A's list, $p(i, j)$ be the position of x_i (in A's list) at the time of the j^{th} access to x_i, and $p(i, k_i + 1)$ be the position of x_i (in A's list) at the end of τ. We may assume that $p(1, 0) < p(2, 0) < \cdots < p(r, 0)$. Let $\beta(i, j)$ be the number of exchanges made during the j^{th} phase for x_i which involve x_i but no other member of X.

As in Theorem 3.1, $\hat{A}$'s exchanges are constructed by modifying A's exchanges. We do the following:

- Ignore any exchanges made by A that involve exactly one member of X, but duplicate every other exchange A makes, at exactly the same time.
- Immediately after servicing σ_1 perform paid exchanges which move the members of X, in an order preserving manner, to be the first r items in the list.
- Immediately after the last access in τ perform paid exchanges which move the members of X, in an order preserving manner, from the first r places in $\hat{A}$'s list to their final positions in A's list (namely, positions $p(1, k_i + 1)$,$p(2, k_i + 1)$,...,$p(r, k_i + 1)$).

A simple induction shows that every exchange in the list constructed above can be performed by $\hat{A}$. It can also be shown that the relative order of all members of X is always the same in A's list and in $\hat{A}$'s list. Let $q(i, 0)$ be the position of x_i in $\hat{A}$'s list after the prepended moves are made, $q(i, j)$ be the position of x_i (in $\hat{A}$'s list) at the time of the j^{th} access to x_i, and $q(i, k_i + 1)$ be the position of x_i (in $\hat{A}$'s list) at the end of τ, before the appended moves are made. Our assumption on the $p(i, 0)$ values gives $q(i, 0) = i$. We will write $\Delta(i, j)$ for $p(i, j) - q(i, j)$. This is how much more A pays than $\hat{A}$ for the j^{th} access to x_i. Notice that for all i and j, $\Delta(i, j) \geq 0$.

Claim 3.4 *For $0 \leq j \leq k_i$ and $1 \leq i \leq r$,*

$$|\Delta(i, j) - \Delta(i, j + 1)| \leq \beta(i, j + 1)$$

Proof The only kind of move which can change the value of $p(i,j)-q(i,j)$ is a move involving exactly one member of X (namely, x_i) since for any other move made by A, $\hat{A}$ makes a move which will preserve $p(i,j)-q(i,j)$. The claim follows immediately.

On the j^{th} access to x_i, A pays $\Delta(i,j)$ more than $\hat{A}$. During the j^{th} phase for x_i, A makes $\beta(i,j)$ paid exchanges which $\hat{A}$ doesn't make. Also, $\hat{A}$ must pay $\sum_{i=1}^{r}\Delta(i,0)$ for the prepended exchanges, and $\sum_{i=1}^{r}\Delta(i,k_i+1)$ for the appended exchanges.

Now we proceed much as in Theorem 3.1. We have that

$$\begin{aligned}
A(\tau)-\hat{A}(\tau) &= \sum_{i=1}^{r}\sum_{j=1}^{k_i+1}\beta(i,j)+\sum_{i=1}^{r}\sum_{j=1}^{k_i}\Delta(i,j)-\sum_{i=1}^{r}\Delta(i,0)-\sum_{i=1}^{r}\Delta(i,k_i+1)\\
&= \sum_{i=1}^{r}\sum_{j=2}^{k_i}\beta(i,j)+\sum_{i=1}^{r}\sum_{j=2}^{k_i-1}\Delta(i,j)\\
&\quad +\sum_{i=1}^{r}(\beta(i,1)+\Delta(i,1)-\Delta(i,0))\\
&\quad +\sum_{i=1}^{r}(\beta(i,k_i+1)+\Delta(i,k_i)-\Delta(i,k_i+1))
\end{aligned}$$

The last two sums are non-negative by Claim 3.4, and the first two sums are clearly non-negative, so $A(\tau)\geq\hat{A}(\tau)$. ∎

3.2 The Eager Move Theorem

The next theorem shows that sometimes it is possible to move an item to the front in an eager fashion. If between accesses to item x no other item preceding x in the list is accessed twice, and if we know that we are going to move x on its second access, then we may as well move x on its first access.

Theorem 3.5 *Let R be a request sequence of the form $xy_1y_2\cdots y_kx\sigma$ where x is any item, the y_i are distinct items, all different from x, and σ any request sequence. For every algorithm A for servicing R, if A moves x to the front before servicing σ, there is an algorithm $\hat{A}$ which moves x to the front on the first access to it, such that the cost to $\hat{A}$ on R is no more than the cost to A on R.*

Proof Once again, the strategy will be to have $\hat{A}$ move x to the front and then mimic A's moves; the move is paid for by the first access. Since A moves x forward, there is no return trip to pay for. The only other costs to worry about are the accesses to the y_i's. They will be paid for by A's moving x to the front. As usual, we assume that A makes no free exchanges.

Consider the sequence of exchanges made by A between the first access to x and the first access in σ. We modify this sequence to obtain $\hat{A}$'s sequence of exchanges by deleting any exchanges which involve x, and prepending exchanges which move x to the front. As before, it is easy to verify that the sequence can be performed.

Suppose that there are β exchanges made by A which involve x. Also, for $1 \leq i \leq k$, let b_i be 1 if at the time of the access to y_i, x is behind y_i in A's list, and let b_i be 0 otherwise. For the access to y_i, $\hat{A}$ pays b_i more than A. So, $A(xy_1y_2\cdots y_kx) - \hat{A}(xy_1y_2\cdots y_kx) \geq \beta - \sum_{i=1}^{k} b_i$. If $b_i = 1$, A must make at least one paid exchange involving x to move x in front of y_i. Since each exchange can only move x in front of one y_i, and the y_i's are distinct, we must have that $\beta \geq \sum_{i=1}^{k} b_i$, so $A(xy_1y_2\cdots y_kx) \geq \hat{A}(xy_1y_2\cdots y_kx)$. ∎

Notice that the theorem can be applied iteratively. For example, consider request sequence $\sigma = \langle 3,2,1,3,2,1,3,2,1,3,3\rangle$. By the Long Block Theorem, an algorithm for servicing σ may as well move 3 to the front on its fourth access (the second to last request). By repeatedly applying the above theorem, the algorithm may as well move 3 to the front on its third second and first accesses.

3.3 Subset Transfers

Between each pair of accesses there are, *a priori*, $n!$ ways for a list update algorithm to rearrange its list. The Subset Transfer Theorem shows that in designing an optimum algorithm it is possible to restrict our attention to at most 2^n of these. The Subset Transfer Theorem holds in the standard model and in any P^d model (only paid exchanges, at cost d), so it is our most general theorem.

Recall that an inversion between two lists is a pair of items (x, y), such that x is in front of y in one list but behind y in the other. Given two lists L_1 and L_2, we write $\text{inv}(L_1, L_2)$ for the set of inversions between the two lists. The next lemma is useful in computing the cost of rearranging a list.

Lemma 3.6 *The number of inversions between two lists equals the minimum number of paid exchanges needed to convert one list to the other.*

Proof Since any exchange either creates one inversion or destroys one inversion, it suffices to show that if two lists have k inversions, then it is possible to get from one to the other using k exchanges.

We use induction on the length of the lists. When both lists are empty the result is obvious. Suppose the lists, L_1 and L_2, are non-empty, that the first item in L_2 is x, and that x is at position p in L_1. If we do $p-1$ paid exchanges to move x to the front of L_1, getting list L_1', then there are $p-1$ fewer inversions between L_1' and L_2 than between L_1 and L_2. Now delete item x from L_1' and L_2, and use induction on the resulting lists. ∎

Suppose item x is accessed. The sequence of paid exchanges made just prior to the access are called a *subset transfer* if they move some subset of the items preceding x to just behind x in such a way that the ordering of items in the subset among themselves remains unchanged.

Theorem 3.7 (Subset Transfer Theorem) *There is an optimum off-line algorithm that does only subset transfers.*

Proof Let A be any algorithm which makes no free exchanges. Suppose that the first request is an access to item x. Consider how A's list changes due to exchanges made before the access. Let R be the items that start in front of x and end up in front of x at the time the access is performed; let S be the items that start in front of x but end up behind x; and let T be the items that start behind x and end up in front of x. Finally, let r, s, and t be the sizes of R, S, and T, respectively. Suppose that up to the access to x, A makes p_0 paid exchanges. Then, since A pays $r+t+1$ for the access, A incurs a total cost of $p_0+r+t+1$.

Consider an algorithm $\hat{A}$, that, given the same request sequence and initial list as A, does the following:

Phase 1: $\hat{A}$ makes the minimal number of paid exchanges, say p_1, to perform a subset transfer on the items in S, and then pays for the access,

Phase 2: $\hat{A}$ makes the minimal number of paid exchanges, say p_2, to make the list look like A's list does at the time when A pays for the access.

The total cost to $\hat{A}$ for both phases is p_1+p_2+r+1. We will show that $p_0 \geq p_1+p_2$, which implies that the cost to $\hat{A}$ is no more than the cost to A.

Let L_0 be $\hat{A}$'s list before phase 1, L_1 be $\hat{A}$'s list after phase one, and L_2 be $\hat{A}$'s list (also A's list) after phase 2. Any inversion between L_0 and L_1 must involve an element of S and either x or an element of R. No inversion between L_1 and L_2 can be of this form, so $\mathrm{inv}(L_0,L_1) \cap \mathrm{inv}(L_1,L_2) = \emptyset$. It follows from this that $\mathrm{inv}(L_0,L_1)$ and $\mathrm{inv}(L_1,L_2)$ partition $\mathrm{inv}(L_0,L_2)$. Therefore, by Lemma 3.6,

$$\begin{aligned} p_0 &\geq |\mathrm{inv}(L_0,L_2)| \\ &= |\mathrm{inv}(L_0,L_1)| + |\mathrm{inv}(L_1,L_2)| \\ &= p_1+p_2. \end{aligned}$$

We have constructed an algorithm which makes a subset transfer on the first access followed by some sequence of paid exchanges. By induction we can construct an algorithm which makes subset transfers on each access, and after the last access makes some sequence of paid exchanges. These paid exchanges can be eliminated since they occur after the last access. This gives an algorithm which makes only subset transfers whose cost is no more than the cost of the original algorithm. ∎

The above proof clearly holds for any algorithm in the P^d models.

4 Implementation of an Optimum Algorithm

An instance of the list update problem consists of a list L and a request sequence σ. We assume σ contains only access requests. Let n be the length of the list and m be

the number of accesses in σ. In general, we do not know how to compute an optimum way to service σ in time polynomial in n and m. However, when $n = 2$ there is a particularly simple algorithm.

Proposition 4.1 *The following algorithm for maintaining a 2 item list achieves the optimum cost on every request sequence: Make no change to the list unless the next two requests are to the last item in the list. In this case, move the requested item to the front via free exchanges after the first request to it.*

Proof Let $A(\sigma)$ be the cost of servicing request sequence σ according to the rule above. We use induction on the length of the request sequence to show that $A(\sigma) = \mathrm{OPT}(\sigma)$. Suppose the list is $1, 2$. By the Long Block and Subset Transfer Theorems, it suffices to consider the induction step for request sequences of the form 21σ (it is possible to avoid using these theorems, by also considering the cases 11σ, 12σ, and 22σ). Let $\mathrm{OPT}(\sigma|1,2)$ and $\mathrm{OPT}(\sigma|2,1)$ be the optimum cost of servicing σ given that the list is $1, 2$ or $2, 1$, respectively. It is easy to verify that $\mathrm{OPT}(21\sigma) = \min\{3 + \mathrm{OPT}(\sigma|1,2), 4 + \mathrm{OPT}(\sigma|2,1)\}$. Since $\mathrm{OPT}(\sigma|1,2) \leq 1 + \mathrm{OPT}(\sigma|2,1)$, we have that $\mathrm{OPT}(21\sigma) = 3 + \mathrm{OPT}(\sigma|1,2)$. Since, by induction, $A(21\sigma) = 3 + A(\sigma) = 3 + \mathrm{OPT}(\sigma|1,2)$, we are done. ∎

Notice that the above algorithm does not need to know *all* the future requests, only the current request and the next request. That is, it uses only 2 look ahead. When the list has more than two items, however, no fixed amount of look ahead suffices for an optimum algorithm.

Theorem 4.2 *No algorithm for a 3 item list with only k look ahead is an optimum list update algorithm.*

Proof Suppose A is a k look ahead algorithm for maintaining a 3 item list. In the proof of Theorem 3.7, the transformation of an algorithm to one which makes only subset transfers preserves the amount of look ahead needed by the algorithm. We can therefore assume that A's only moves are subset transfers. We may also assume that $k \geq 3$.

Suppose A's list is $1, 2, 3$, and consider the request sequences $\sigma_1 = 31^{k-1}2$ and $\sigma_2 = 31^{k-1}3$. Since A has only k look ahead, the first subset transfer it makes must be the same whether it services σ_1 or σ_2. Consider what order A's list could be in after servicing the first k requests of σ_1 or σ_2. Table 1 shows for each possible first move the positions reachable and the minimum cost for reaching it. (The entries are easily verified by induction on k.) Notice that for A to achieve the optimum cost on either σ_1 or σ_2, A's first move must either be to do nothing or to switch the last two items (the first two moves of Table 1). If A does nothing, then it cannot be optimum on σ_2, while if it makes the second move it cannot be optimum on σ_1. ∎

It is possible to construct more interesting request sequences which require a lot of look ahead. A tedious but straight forward calculation shows that no algorithm

First Move	Position	Cost
$1,2,3 \rightarrow 1,2,3$	$1,2,3$	$k+2$
$1,2,3 \rightarrow 1,3,2$	$1,3,2$	$k+2$
$1,2,3 \rightarrow 2,3,1$	$1,2,3$	$k+5$
	$1,3,2$	$k+7$
	$2,1,3$	$2k+3$
	$3,1,2$	$2k+4$
	$2,3,1$	$3k+1$
$1,2,3 \rightarrow 3,1,2$	$1,3,2$	$k+3$
	$3,1,2$	$2k+1$

Table 1: Positions reachable after servicing 31^{k-1} for $k \geq 3$.

with only $3k+2$ look ahead can achieve the optimum cost on a request sequence of the form $32(312)^k\sigma$.

In [6], Manasse, McGeoch, and Sleator discuss *task systems*, of which list update is a special case. They give a dynamic programming algorithm for finding the minimal cost to service a request sequence in any task system. When applied to list update this works as follows: Suppose the request sequence σ is of length m. Let $\mathrm{DYN}(l,i)$ be the optimum cost to service the first i requests of σ and end with the list in order l. Here $i \in \{1,2,\ldots,m\}$, and l is one of the $n!$ permutations of n items. Computing $\mathrm{DYN}(l,0)$, for all l, is quite easy. Given $\mathrm{DYN}(l,i)$ for all l, we can compute $\mathrm{DYN}(l,i+1)$ by considering all possible ways to service the $(i+1)^{\mathrm{st}}$ request and change the list from some order l to another order l'. $\mathrm{OPT}(\sigma)$ is the minimum over all l of $\mathrm{DYN}(l,m)$. Computing $\mathrm{DYN}(l,i)$ for all l and i takes time $\Theta((n!)^2 m)$. The space requirement is $\Theta(n!)$ if we desire only the cost and $\Theta(mn!)$ if we desire the actual moves as well.

We can improve the running time substantially by using the Subset Transfer Theorem (Theorem 3.7) to reduce the number of transitions that must be examined. Consider the i^{th} access in the request sequence. There are $(n-1)!$ permutations of the list in which the accessed item is at position k. If the item is at position k there are 2^{k-1} subsets to check. Summing over all k, we find that the time to process the i^{th} request is $O(m2^n(n-1)!)$. This is currently the best running time of any algorithm which computes an optimum algorithm.

Theorem 4.3 *There is an algorithm which computes the optimum cost for servicing m accesses to an n item list which runs in time $O(m2^n(n-1)!)$.*

We can use the Long Block Theorem and its generalization (Theorems 3.1 and 3.3) to reduce the number of transitions to examine even further. We preprocess the request sequence and require that the algorithm moves the requested item to the front on long blocks, *i.e.*, when it is requested twice or more consecutively. Theorems 3.1 and 3.3 ensure that an optimum algorithm with this behavior exists. Since repeated

references are common both in random data and in typical sample data, this will often give a significant improvement in running time. Similarly, one could also use the Eager Move Theorem to reduce the running time.

We have implemented the dynamic programming algorithm with the subset transfer and long block improvements described above for use in our research on on-line algorithms for list update. It runs reasonably well for very small values of n, say at most 6, but begins to slow appreciably for n larger than 6. At $n = 7$, for example, there are approximately ninety-two thousand probes of the DYN(l, i) table per request. For comparison, at $n = 7$ the naive implementation of the dynamic programming algorithm does approximately twenty-five million probes per request.

It is easy to see that any optimum off-line algorithm that uses the subset transfer theorem for sequences of accesses can be extended to one for sequences of accesses and insertions. Suppose the insertion operations are all moved in order to the beginning of the request sequence. The optimum cost of processing this sequence is the same as that of the original sequence. Handling deletions is somewhat more complicated. Our dynamic programming algorithm can be extended to deletions in two ways. One possiblity is to simply leave deleted elements in the list, but modify the cost function so that they add nothing to the cost of either accesses or subset transfers. A second possibility is to modify the algorithm to simulate a deletion by moving the deleted element to the back of the list, at zero cost. This saves some space, since deleted elements may be reused for subsequently inserted items. In either case, the optimum cost computed is the same.

5 Remarks and Open Problems

Our work in this paper characterizes the nature of optimum algorithms for the list update problem. In the Long Block Theorem, we show that the optimum algorithm *must* move an item to the front of the list if it is accessed three or more times in a row. Furthermore, it pays to move sub-collections of items to the front quickly when those items are accessed much more often than other items. Furthermore, failure to follow the Long Block Theorem can be very expensive. The move-to-front heuristic always abides by the Long Block Theorem, and generally simulates the generalization of the Long Block Theorem and Eager Move Theorems.

We are also able to use the Subset Transfer Theorem to reduce significantly the computation time of the dynamic programming algorithm. We leave open, however, the problem of finding an algorithm that is polynomial in both n and m, or conversely showing that such an algorithm is unlikely or impossible. It is possible that our characterizations give information about the optimum algorithm which can be used to derive a lower bound for computing an optimal solution. Another interesting question is whether there is a polynomial time algorithm that computes a near-optimal solution.

Acknowledgements

We thank Bill Gasarch for his proofreading and helpful suggestions.

References

[1] J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Commun. ACM*, 28(4):404–411, Apr. 1985.

[2] D. L. Black and D. D. Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[3] P. J. Burville and J. F. C. Kingman. On a model for storage and search. *Journal of Applied Probability*, 10:697–701, 1973.

[4] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New results on server problems. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, pages 291–300, 1990.

[5] W. J. Hendricks. An account of self-organizing systems. *SIAM J. Comput.*, 5(4):715–723, Dec. 1976.

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

[7] J. McCabe. On serial files with relocatable records. *Operations Research*, 13:609–618, 1965.

[8] N. Reingold and J. Westbrook. Randomized algorithms for the list update problem. Technical Report YALEU/DCS/TR-804, Yale University, 1990.

[9] R. Rivest. On self-organizing sequential search heuristics. *Commun. ACM*, 19(2):63–67, February 1976.

[10] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.