The List Update Problem

Erez Timnat
The List Update Problem

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Erez Timnat

Submitted to the Senate
of the Technion — Israel Institute of Technology
Nisan 5776 Haifa May 2016
This research was carried out under the supervision of Prof. Seffi Naor, in the Faculty of Computer Science.

The generous financial help of the Technion is gratefully acknowledged.
## Contents

Abstract

1 Introduction
1.1 Online Computation and Competitive Analysis 
1.2 The List Update Problem
1.2.1 Algorithms for List Update

2 List Update Algorithms via Linear Programming
2.1 The Linear Program
2.1.1 The Variables
2.1.2 The Objective Function
2.1.3 The Constraints
2.1.4 Formulation Limitations
2.2 The Dual Program
2.3 Analyzing MTF
2.3.1 Assigning Values to Primal Variables
2.3.2 Assigning Values to Dual Variables
2.3.3 Competitiveness
2.4 Analyzing TIMESTAMP, BIT and COMB
2.4.1 The Phase Partitioning Technique
2.4.2 The Dual Assignment
2.4.3 Proving Dual Feasibility
2.4.4 Value of Dual Solution
2.4.5 Obtaining the Competitiveness of TS, BIT, And COMB

3 Lower Bound For Free Exchanges Only
3.1 Defining the Example
3.2 The Cost Of OPT
3.2.1 Calculating the Cost of OPT
3.2.2 OPT is Optimal
3.3 The Cost Of OPT_FREE
3.3.1 Three Consecutive Requests
3.3.2 Understanding the Optimal Algorithm OPT_FREE ............... 31
3.3.3 Calculating the Cost of OPT_FREE ............................. 31
3.4 Calculating the Maximal Ratio ..................................... 32

Hebrew Abstract

i
Abstract

In this work, we consider the list update problem, originally defined in the seminal work on competitive analysis by Sleator and Tarjan [ST85]. In this problem, a sequence of requests, consisting of items to access in a linked list, is given. After an item is accessed it can be moved to any position forward in the list at no cost (free exchange), and, at any time, any two adjacent items can be swapped at a cost of 1 (paid exchange). The cost to access an item is its current position in the list. The goal is to dynamically rearrange the list so as to minimize the total cost incurred from accesses and exchanges.

We present novel proofs for the competitive ratio of the classic algorithms for the list update problem. These proofs are obtained via a novel linear program for the list update problem. We apply the dual fitting technique, to analyze the competitiveness of classic algorithms for the list update problem: MTF, BIT, TIMESTAMP and COMB.

We also present a new and improved lower bound of 5/4 for offline algorithms, that are allowed to use only free exchanges. This is a significant improvement over the previous known lower bound of 12/11 [LORR15]. To achieve this, we present a novel basic gap example. We then analyze the cost of the general optimal algorithm, that is allowed to use paid exchanges, and the cost of an optimal algorithm that uses only free exchanges. We show a gap of 5/4 for our example. By repeating the same basic input sequence over different items, we show that this gap is multiplicative and not additive, since we can repeat it as many times as we want. This result teaches us that in order to achieve a better approximation than 5/4 for offline list update, one must make use of paid exchanges. Note that the classic algorithms - MTF, BIT, TIMESTAMP and COMB - do not use paid exchanges at all.
Chapter 1

Introduction

1.1 Online Computation and Competitive Analysis

In online computation, an algorithm has to operate and make decisions based on past events. It does not know about future events that might be relevant for its current choices. In contrast, an offline algorithm is one that possesses knowledge of all future events as well, and can plan its actions accordingly. An online algorithm is presented with a sequence of requests $\sigma = \sigma_1 \sigma_2 \ldots \sigma_m$; the cost of an online algorithm $A$ for servicing the sequence $\sigma$ is denoted by $A(\sigma)$. In a minimization problem, we wish to minimize $A(\sigma)$. An optimal algorithm $OPT$ for the sequence $\sigma$, is one that minimizes the cost $OPT(\sigma)$. When measuring the performance of an online algorithm, it is often interesting to compare its cost to the cost of an optimal offline algorithm. We say that a deterministic online algorithm is $c$-competitive, if for every sequence of events $\sigma = \sigma_1 \sigma_2 \ldots \sigma_m$ the following holds:

$$A(\sigma) \leq cOPT(\sigma) + b,$$

where $b$ and $c$ are constants, independent of the request sequence. We say that a random online algorithm is $c$-competitive, if for every sequence of events $\sigma$ the following holds:

$$E[A(\sigma)] \leq cOPT(\sigma) + b.$$

The expectation is taken over the random choices of the algorithm, and holds for any possible input sequence $\sigma$.

One of the most well studied problems in the field of online computation and competitive analysis [BEY98] is the paging problem. In this problem, requests are for pages of memory. A request is served by guaranteeing that a requested page $p$ is available in the fast memory. If it is not currently available, then the system must load it from the slow memory into the fast memory, thus incurring a cost of one page fault. When doing so, the system has to decide which page to evict from the fast memory.
to make space for p. In order to minimize the number of page faults, the choice of which page to evict must be made wisely. Different algorithms suggest different eviction policies, resulting in a different number of page faults.

A natural generalization of the paging problem is the \( k \)-server problem. In this problem, we have a metric space, consisting of a set of points \( M \), and a distance metric over this set, \( d \). We have \( k \) servers located in the beginning at different points in the metric space. The algorithm is presented with a set of requests \( \sigma = \sigma_1\sigma_2...\sigma_m \), where each request \( \sigma_i \) is a point in the metric space. We say that a request \( \sigma_i = r \) is served if one of the servers is located at \( r \). By moving servers, the algorithm must serve all requests sequentially. The cost of an algorithm ALG serving a request sequence \( \sigma \) is defined as the total distance (with respect to the metric \( d \)) traveled by ALG’s servers while servicing \( \sigma \). Paging is an instance of the \( k \)-server problem with a uniform metric space (all distances are 1), where the \( k \) servers correspond to the locations in the fast memory, and \( |M| \) is the number of slow memory pages.

1.2 The List Update Problem

A fundamental, well studied, problem in online computation and competitive analysis is the list update problem. We are given a linked list of items with keys and values. We are given requests to items, by their keys, and must serve the requests by accessing the requested item, and returning the value of the item. To find an item in the list, we go over the items from the first one and onwards, and compare the requested key with the keys in the list. Hence, accessing the \( i \)-th item in the list incurs a cost of \( i \) units in the full cost model. In the partial cost model, we say that accessing the \( i \)-th item costs \( i - 1 \). In the partial cost model, the cost is equal to the number of items preceding the requested item in the list, excluding the requested item itself. And so the difference between the two models is of one unit cost per request, and of \( m \) units cost for a request sequence of length \( m \). The goal is to keep access costs small by rearranging the items in the list. There are two possible ways to exchange items: free exchanges, and paid exchanges. After an item has been requested, it may be moved free of charge closer to the front of the list. This is called a free exchange. Any other exchange of two consecutive items in the list incurs unit cost and is called a paid exchange. In online list update, each request must be served immediately upon arrival, without any knowledge of future requests. In offline list update, the algorithm gets the entire sequence of requests in the beginning, and may plan its actions accordingly.

1.2.1 Algorithms for List Update

Several algorithms have been proposed for the online list update problem. The move-to-front rule (MTF), for example, moves each item to the front of the list after it has been requested. Sleator and Tarjan have shown that MTF is 2-competitive [ST85]. This is
also the best possible competitiveness for any deterministic online algorithm for the list update problem \cite{rw90}. Another 2-competitive algorithm is TimeStamp, or TS \cite{alb98}. TS moves the requested item \( x \) in front of all items which have been requested at most once, since the last request to \( x \).

As shown first by Irani \cite{ira91}, randomized algorithms for the list update problem can perform better. An example of such an algorithm is \( BIT \). It initializes each item \( x \) with a random bit \( b(x) \). When a request to access an element \( x \) is given, first complement its bit \( b(x) \). Then, if \( b(x) = 1 \), move \( x \) to the front; otherwise (\( b(x) = 0 \)) do nothing. \( BIT \) was shown to be 1.75-competitive by Reingold and Westbrook. The best randomized list update algorithm known to date is the 1.6-competitive algorithm \( COMB \) \cite{asw95}. It serves the request sequence with probability 0.8 using the algorithm \( BIT \). With probability 0.2 it serves the request sequence using TS.

Lower bounds for the competitive ratio of randomized algorithms are harder to find. The tightest lower bound known for the online list update problem is 1.50115 \cite{amb01} for the partial cost model, and 1.5 \cite{tei93} for the full cost model. The optimal competitive ratio for the online list update problem is therefore between 1.5 and 1.6, but the true value is still unknown.

As for the offline problem, it was proven to be NP-hard by \cite{amb00}. There are several non-polynomial time algorithms for the optimal offline cost, but not much work on the development of approximation algorithms for the offline problem. The online algorithms can also be applied offline, thus providing a 1.6 approximation for the offline problem. In fact, to date, there is no known offline algorithm that guarantees a better approximation factor.
Chapter 2

List Update Algorithms via Linear Programming

In this chapter we present a novel linear program for the list update problem. We then present a novel analysis for the classic algorithms: MTF, TIMESTAMP, BIT, and COMB using our linear program. The analysis uses a technique called dual fitting, which is also presented in this chapter.

2.1 The Linear Program

We think of the requests as coming each at a different time unit. Time starts at \( t = 0 \), in which we are given an initial list state. We denote the length of the request sequence by \( T \). So at every time \( 1 \leq t \leq T \) comes an additional request, \( \sigma_t \). We then make exchanges in the list, and only then serve the request.

We write a linear program that captures only paid exchanges, and not free exchanges. It relies on the following lemma:

Lemma 2.1.1. There exists an optimal offline algorithm \( \text{OPT} \) for the list update problem that uses only paid exchanges.

Proof. We show that any algorithm using free exchanges can be replaced with an algorithm using only paid exchanges, incurring the exact same cost.

Consider an algorithm \( \text{ALG} \), that after accessing the \( i \)-th item, called \( x \), places it in the \( j \)-th position using a free exchange, where \( j < i \). Now consider an algorithm \( \text{ALG1} \) which replaces every free exchange of \( \text{ALG} \) by a series of paid exchanges. Before servicing the request for \( x \), \( \text{ALG1} \) substitutes \( x \) with its predecessor repeatedly, until it reaches position \( j \), and only then - serves the request. This incurs a cost of \( i - j \) for the series of paid exchanges, and a cost of \( j \) for accessing the item afterwards. Thus the total cost is \((i - j) + j = i\), exactly the same as the original algorithm. Note that both algorithms finish at exactly the same state, so \( \text{ALG1} \) can continue following \( \text{ALG} \)’s
moves, only without using free exchanges. Thus, ALG1 incurs the same cost, and uses only paid exchanges.

2.1.1 The Variables

We use variables $x_{i,j,t}$ to denote whether item $i$ precedes item $j$ at the end of time unit $t$, that is - when the request is being served, and after any exchanges that we may have done during time $t$. We use variables $z_{i,j,t}$ to denote whether item $j$ moved ahead of item $i$ during time $t$, as a result of the exchanges done at time $t$.

2.1.2 The Objective Function

The objective of the problem is to minimize the cost of the algorithm. The cost of the algorithm is composed of a cost of one unit for each paid exchange, which is captured by the $z$-variables, and from a cost of $i - 1$ (in the partial cost model) for accessing an item at position $i$. $i - 1$ is exactly the number of items preceding an item in position $i$, so this is captured by the $x$-variables. At time $t$, we are accessing item $\sigma_t$, so we count the number of items preceding it to know the cost of accessing it. Hence, in total, our objective function is:

$$
\min \sum_{i=1}^{l} \sum_{j=1, j \neq i}^{l} \sum_{t=1}^{T} z_{i,j,t} + \sum_{t=1}^{T} \sum_{i=1}^{l} x_{i,\sigma_t,t}.
$$

2.1.3 The Constraints

At time 0, we assume without loss of generality, that the order of the list is: 1, 2, 3, .., $\ell$ - that is - the $i$ - th item is in the $i$ - th position. The constraints for this are:

$$
\forall i \neq j : x_{i,j,0} = \begin{cases} 
1 & \text{if } i < j \\
0 & \text{if } i > j 
\end{cases}
$$

At any other time $t$, either item $i$ is before item $j$, or item $j$ is before $i$ - but not both. In the linear program this constraint is:

$$
\forall i \neq j, \forall t : 1 \leq t \leq T : x_{i,j,t} + x_{j,i,t} = 1.
$$

Now we constrain the $z$-variables to match our definition. We wanted variable $z_{i,j,t}$ to denote whether item $j$ moved ahead of item $i$ during time $t$, as a result of the exchanges done at time $t$. Item $j$ surpassed item $i$ during time $t$ if and only if item $i$ was ahead of item $j$, before time $t$, at $t - 1$, and item $j$ is ahead of $i$ at the end of time $t$. That is, $z_{i,j,t} = 1$ if and only if $x_{i,j,t-1} = 1$ and $x_{j,i,t} = 1$, which happens if and only if $x_{i,j,t-1} = 1$ and $x_{i,j,t} = 0$. This leads us to the constraints:

$$
\forall i \neq j, \forall t : 1 \leq t \leq T : z_{i,j,t} + x_{i,j,t} - x_{i,j,t-1} \geq 0
$$
To minimize our objective function, the LP minimizes the \( z \)-variables, and only increases them to 1 if needed. That is, \( z_{i,j,t} = 1 \) if and only if \( x_{i,j,t-1} = 1 \) and \( x_{i,j,t} = 0 \), as defined.

And, of course, we also require that:

\[
x, z \geq 0
\]

### 2.1.4 Formulation Limitations

Any solution to the list update problem naturally leads to a feasible solution of the same value to the linear program. However, the converse is not necessarily true. A solution to the linear program might not be integral, and thus might not lead to a feasible solution for the list update problem. Moreover, even an integral solution might not suggest a legal ordering of the items, because it does not enforce transitivity. For instance, we might get \( x_{i,j,t} = x_{j,k,t} = x_{k,i,t} = 1 \). This implies that at the end of time \( t \) we should have item \( i \) ahead of item \( j \), item \( j \) ahead of item \( k \), and item \( k \) ahead of item \( i \). Of course, this is not a legal ordering of the items. We could add more constraints to enforce transitivity, but we refrain from doing so, as this will not be of help for us.

### 2.2 The Dual Program

Let us write the primal linear program we obtained from the previous section. We use the dual variables \( a_{i,j,t}, b_{i,j,t} \) and \( s_{i,j} \). We show below the primal linear program, writing next to each constraint its corresponding dual variable. This looks as follows:

\[
\min \sum_{i=1}^{l} \sum_{j: j \neq i}^{l} \sum_{t=1}^{T} z_{i,j,t} + \sum_{i=1}^{l} \sum_{t=1}^{T} x_{i,\sigma_t,t}
\]

\[s.t.
\]

\[
\forall i \neq j : x_{i,j,0} = \begin{cases} 
1 & \text{if } i < j /s_{i,j} \\
0 & \text{if } i > j /s_{i,j}
\end{cases}
\]

\[
\forall i \neq j, \forall t : 1 \leq t \leq T : x_{i,j,t} + x_{j,i,t} = 1 /a_{i,j,t}
\]

\[
\forall i \neq j, \forall t : 1 \leq t \leq T : z_{i,j,t} + x_{i,j,t} - x_{i,j,t-1} \geq 0 /b_{i,j,t}
\]

\[
x, z \geq 0
\]

From this, we obtain the following dual linear program:
\[
\max \sum_{i=1}^{l} \sum_{j=1, j \neq i}^{l} \sum_{t=1}^{T} a_{i,j,t} + \sum_{i=1}^{l-1} \sum_{j=i+1}^{l} s_{i,j}
\]

s.t.

For the \(z\)-variables:

\[\forall i \neq j, \forall t : 1 \leq t \leq T : b_{i,j,t} \leq 1\]

For the \(x\)-variables, at times \(1 \leq t \leq T - 1\):

\[\forall i \neq j, \forall t : 1 \leq t \leq T - 1, j = \sigma_t : a_{i,j,t} + a_{j,i,t} + b_{i,j,t} - b_{i,j,t+1} \leq 1\]

\[\forall i \neq j, \forall t : 1 \leq t \leq T - 1, j \neq \sigma_t : a_{i,j,t} + a_{j,i,t} + b_{i,j,t} - b_{i,j,t+1} \leq 0\]

For the \(x\)-variables at time \(t = T\):

\[\forall i \neq j, j = \sigma_T : a_{i,j,T} + a_{j,i,T} + b_{i,j,T} \leq 1\]

\[\forall i \neq j, j \neq \sigma_T : a_{i,j,T} + a_{j,i,T} + b_{i,j,T} \leq 0\]

For the \(x\)-variables at time \(t = 0\):

\[\forall i \neq j, : s_{i,j} - b_{i,j,1} \leq 0\]

And finally:

\[b \geq 0\]

### 2.3 Analyzing MTF

In this section, we present a novel proof that shows that algorithm MTF is 2-competitive. The result itself is known, and was proved by Sleator and Tarjan [ST85]. But Sleator and Tarjan used a potential function for the proof. We prove this using a technique called dual fitting. Proofs with potential functions are usually considered to be some sort of black magic: the potential function seem to come out of no where, and from reading the proof one usually does not gain any intuition about the algorithm or its competitiveness. With dual fitting, the proofs tend to be more straight-forward, and less magical, and so they also teach us more about the nature of the algorithm.

To use dual fitting - we assign the variables of the primal program in accordance with the MTF algorithm, and we also assign values to the dual variables. The dual solution must be feasible, and also, we need to make sure that the value of the primal
objective function is at most twice the value of the dual objective function. After achieving both these goals, we know from weak duality that any value obtained by the dual objective function is a lower bound to any value obtained by the primal objective function. Since we know any value obtained by \( OPT \) can also be obtained by the primal objective function - we know that the value of the dual is a lower bound on the value of \( OPT \). Hence, if the value of the primal solution is upper bounded by twice the dual solution, then it is also upper bounded by twice the value of \( OPT \), which concludes our proof.

Instead of using free exchanges, as \( MTF \) is defined, we redefine it to use only paid exchanges. Upon a request for an item \( x \), we first move \( x \) to the front using paid exchanges, and then access it. If item \( x \) was at position \( i \), then accessing it would cost \( i - 1 \) in the partial cost model, and then moving it to the front is done for free. In our alternative version of \( MTF \), we move \( x \) to the front of the list using \( i - 1 \) paid exchanges, and then we access it when it is at position 1, for a cost of \( 1 - 1 = 0 \). Either way, the total cost is \( i - 1 \), and the list ends up in the same state. Therefore, the two algorithms are equivalent.

### 2.3.1 Assigning Values to Primal Variables

We assign values to the primal variables that imitate the way algorithm \( MTF \) proceeds, and thus the objective function retains the same value as \( MTF \). Specifically, variable \( x_{i,j,t} \) is assigned 1 if item \( i \) is ahead of item \( j \) in the list at the end of time \( t \). Otherwise, it is assigned 0. Variable \( z_{i,j,t} \) is assigned 1 if item \( j \) surpassed item \( i \) in the list during time \( t \). Otherwise, it is assigned 0. Clearly this assignment matches the algorithm, and it is a feasible solution to the primal linear program, that retains the same value as \( MTF \).

### 2.3.2 Assigning Values to Dual Variables

We wish to construct a dual solution that is both feasible (i.e. meets all the constraints), and achieves a value of at least half of the primal solution.

To do this, we assign values presented below, and later show that they indeed satisfy the two stated goals. The assignment is as follows:

The \( b \)-variables get half the previous value of the \( x \)-variables:

\[
\forall i \neq j, \forall t : 1 \leq t \leq T : b_{i,j,t} = \frac{1}{2} x_{i,j,t-1}.
\]

The \( a \)-variables of times \( t \leq T - 1 \) get half the current value of the \( z \)-variables:

\[
\forall i \neq j, \forall t : 1 \leq t \leq T - 1 : a_{i,j,t} = \frac{1}{2} z_{i,j,t}.
\]

The \( a \)-variables of time \( t = T \) get a slightly different value:
∀i \neq j : a_{i,j,T} = \frac{1}{2}z_{i,j,T} - \frac{1}{4}.

And the s-variables get values representing the initial list state:

\[ s_{i,j} = \frac{1}{2}x_{i,j,0}. \]

### 2.3.3 Competitiveness

First, since we used a primal assignment based on MTF, we know our primal solution is feasible. Second, we need to show that the ratio between the primal and dual solutions is exactly 2. Third, we need to show that the dual solution is feasible. This will conclude our proof.

#### Calculating the Ratio

We have redefined the MTF algorithm so that its cost is entirely from paid exchanges, since it only accesses items after they are at the front of the list. Therefore,

\[ P = \sum_{i=1}^{l} \sum_{j=1}^{l} \sum_{j \neq i}^{l} \sum_{t=1}^{T} z_{i,j,t}. \]

The dual objective function is:

\[ D = \sum_{i=1}^{l} \sum_{j=1}^{l} \sum_{j \neq i}^{l} \sum_{t=1}^{T} a_{i,j,t} + \sum_{i=1}^{l-1} \sum_{j=i+1}^{l} s_{i,j} = \sum_{i=1}^{l} \sum_{j=1}^{l} \sum_{j \neq i}^{l} \left( \sum_{t=1}^{T-1} \frac{1}{2}z_{i,j,t} + \frac{1}{2}z_{i,j,T} - \frac{1}{4} \right) + \sum_{i=1}^{l-1} \sum_{j=i+1}^{l} \frac{1}{2}x_{i,j,0} = \]

\[ \sum_{i=1}^{l} \sum_{j=1}^{l} \sum_{j \neq i}^{l} \sum_{t=1}^{T} \frac{1}{2}z_{i,j,t} - \sum_{i=1}^{l-1} \sum_{j=i+1}^{l} \frac{1}{4} + \sum_{i=1}^{l-1} \sum_{j=i+1}^{l} \frac{1}{2}x_{i,j,0}. \]

We have simply replaced the dual variables by their assignment as defined in the previous subsection. The second sum goes over all pairs \( i \neq j \) twice. The third sum is only over \( i < j \), for which \( x_{i,j,0} = 1 \). Therefore, we obtain the following:

\[ D = \sum_{i=1}^{l} \sum_{j=1}^{l} \sum_{j \neq i}^{l} \sum_{t=1}^{T} \frac{1}{2}z_{i,j,t} - \sum_{i=1}^{l-1} \sum_{j=i+1}^{l} \frac{1}{2} + \sum_{i=1}^{l-1} \sum_{j=i+1}^{l} \frac{1}{2} = \sum_{i=1}^{l} \sum_{j=1}^{l} \sum_{j \neq i}^{l} \sum_{t=1}^{T} \frac{1}{2}z_{i,j,t} = \frac{1}{2}P. \]

And so the ratio is exactly 2.

#### Dual Feasibility

We simply go over the different constraints one by one, and show they are all satisfied.

The first set of constraints is:
∀i ≠ j, ∀t : 1 ≤ t ≤ T : b_{i,j,t} ≤ 1.

No \( b \)-variable ever gets higher than \( \frac{1}{2} \), so these constraints clearly hold.

The next set of constraints is:

\[ \forall i \neq j, \forall t : 1 \leq t \leq T - 1, j = \sigma_t : a_{i,j,t} + a_{j,i,t} + b_{i,j,t} - b_{i,j,t+1} \leq 1. \]

Since \( j = \sigma_t \), we are now accessing item \( j \) and moving it to the front, so \( j \) might surpass \( i \), but \( i \) surely does not surpass \( j \), and thus: \( a_{j,i,t} = 0 \).

Additionally, \( a_{i,j,t} \leq \frac{1}{2} \); \( b_{i,j,t} \leq \frac{1}{2} \), and thus:

\[ a_{i,j,t} + a_{j,i,t} + b_{i,j,t} - b_{i,j,t+1} \leq \frac{1}{2} + 0 + \frac{1}{2} - 0 = 1, \]

and the constraint holds. The next set of constraints:

\[ \forall i \neq j, \forall t : 1 \leq t \leq T - 1, j \neq \sigma_t : a_{i,j,t} + a_{j,i,t} + b_{i,j,t} - b_{i,j,t+1} \leq 0. \]

Now \( j \neq \sigma_t \). If items \( i \) and \( j \) do not switch places at time \( t \), then \( a_{i,j,t} = a_{j,i,t} = 0 \) and \( b_{i,j,t} - b_{i,j,t+1} = 0 \), so the constraint holds.

Otherwise, they do switch places, and since \( j \neq \sigma_t \), we are not accessing \( j \), so we must be accessing \( i \). If they switched places, that means \( i \) surpassed \( j \), which means: \( z_{j,i,t} = 1 \); \( z_{i,j,t} = 0 \); \( x_{i,j,t-1} = 0 \); \( x_{i,j,t} = 1 \). Therefore:

\[ a_{i,j,t} + a_{j,i,t} + b_{i,j,t} - b_{i,j,t+1} = \frac{1}{2}(z_{i,j,t} + z_{j,i,t} + x_{i,j,t-1} - x_{i,j,t}) = \frac{1}{2}(0 + 0 + 1) = 0. \]

And so the constraint holds. The next set of constraints is for time \( T \):

\[ \forall i \neq j, j = \sigma_T : a_{i,j,T} + a_{j,i,T} + b_{i,j,T} \leq 1. \]

\[ \forall i \neq j, j \neq \sigma_T : a_{i,j,T} + a_{j,i,T} + b_{i,j,T} \leq 0. \]

Substituting the variables with their assigned values we obtain:

\[ a_{i,j,T} + a_{j,i,T} + b_{i,j,T} = \frac{1}{2}z_{i,j,T} - \frac{1}{4} + \frac{1}{2}z_{j,i,T} - \frac{1}{4} + \frac{1}{2}x_{i,j,T-1} = \frac{1}{2}(z_{i,j,T} + z_{j,i,T} + x_{i,j,T-1}). \]

For \( j = \sigma_T \), we are moving item \( j \) to the front, and thus: \( \frac{1}{2}z_{j,i,T} = 0 \), since item \( i \) did not surpass \( j \), since we are now accessing \( j \). Therefore, we are left with:

\[ \frac{1}{2}(z_{i,j,T} + z_{j,i,T} + x_{i,j,T-1}) \leq \frac{1}{2}(1 + 0 + 1 - 1) = \frac{1}{2} \leq 1, \]

so this constraint clearly holds. For \( j \neq \sigma_T \):

If items \( i \) and \( j \) did not switch places, then \( z_{i,j,t} = z_{j,i,t} = 0 \), and thus: \( \frac{1}{2}(z_{i,j,T} + z_{j,i,T} + x_{i,j,T-1}) \leq \frac{1}{2}(0 + 0 + 1) = 0 \) and the constraint holds.
Otherwise, the items can only switch places if we are accessing item $i$, since we know that $\sigma_i \neq j$, so we are not accessing $j$. Then, $z_{i,j,T} = 0$; $z_{j,i,T} = 1$. If $i$ surpasses $j$ at time $T$, that means that $i$ was not ahead of $j$ at time $T - 1$, and thus $x_{i,j,T-1} = 0$. So we obtain:

$$\frac{1}{2}(z_{i,j,T} + z_{j,i,T} + x_{i,j,T-1} - 1) = \frac{1}{2}(0 + 1 + 0 - 1) = 0,$$

and the constraint holds. The last set of constraints is for time $t = 0$:

$$\forall i \neq j : s_{i,j} - b_{i,j,1} \leq 0.$$

But we have defined $s_{i,j} - b_{i,j,1} = \frac{1}{2}x_{i,j,0} - \frac{1}{2}x_{i,j,0} = 0$, so these constraints clearly hold.

### 2.4 Analyzing TIMESTAMP, BIT and COMB

In the following section we analyze the competitiveness of three known algorithms for the online list update problem.

We use the phase partitioning technique to analyze the three algorithms, similarly to their original analysis [BEY98]. However, instead of analyzing the performance of $OPT$ directly, we use the dual linear program to obtain a lower bound for $OPT$.

By analyzing the cost of each algorithm in every phase, and analyzing the cost of the dual solution in every phase, we obtain the desired competitive ratios.

#### 2.4.1 The Phase Partitioning Technique

The notation $\sigma_{xy}$ is used for the projection of a sequence of requests $\sigma$ on a pair of items $x,y$, which is done by removing requests for all other items. For instance, if $\sigma = <x,y,z,x,z,y>$, then $\sigma_{xy} = <x,y,x,y>$. An algorithm is called projective if for any pair of items $x,y$ the relative order of the two items $x,y$ depends solely on their projection, $\sigma_{xy}$, and not on requests for other items. That is, the relative order of $x,y$ in the list does not change when there are requests to other items. Moreover, during requests for the items $x,y$ themselves - they change, or not change, their relative order regardless of the number of items between them, ahead of them, or after them in the list. An algorithm that satisfies this property for any input sequence $\sigma$ and for any pair of items $x,y$ is called a projective algorithm.

The three algorithms $BIT$, $TS$, and $COMB$ are all projective [BEY98]. For analyzing the competitive ratio of projective algorithms in the partial cost model, it is sufficient to analyze it for a list of two items only. This is known as the factoring lemma [BEY98].

**Lemma 2.4.1.** Factoring Lemma.

Let $alg$ be a projective online list accessing algorithm. Suppose that for every pair $\{x,y\} \subseteq L$, and for every request sequence $\sigma$, $alg(\sigma_{xy}) \leq cOPT(\sigma_{xy})$. Then $alg$ is
In the phase partitioning technique, we consider a pair of items \( \{x, y\} \subseteq L \). We partition \( \sigma_{xy} \) into phases, each of which terminates with two consecutive requests for the same item. We look at three types of phases when \( x \) starts ahead of \( y \), and symmetrically, there are three types of phases, when \( y \) starts ahead of \( x \). Note that all three algorithms finish the three phases with the same list order, since when accessing the same item twice consecutively - each algorithm would have this item at the front of the list. Since each phase ends with two consecutive requests for the same item - all three algorithms finish each phase with the same relative order between \( x \) and \( y \). The three types of phases when \( x \) starts ahead of \( y \) are:

(a) \( x^iyy \)
(b) \( x^i(yx)^kyy \)
(c) \( x^i(yx)^kx \)

where \( i, k \) are integers satisfying \( i \geq 0 \) and \( k \geq 1 \).

The request sequence might end with requests that are only a beginning of a phase, rather than a full phase. However, such requests can be completed into a full phase by adding at most two requests, which add only an additive constant of up to 2 to each algorithm, for every such pair. We have \( \binom{1}{2} \) pairs in the list, and hence an additive factor of at most \( 2\binom{1}{2} \) for the full request sequence. Therefore, for analyzing the multiplicative factor, which is the competitive ratio, we can simply ignore these requests, since they all can be accounted for with the additive factor.

An analysis shown in [BEY98] teaches us that the cost of the three algorithms for serving each phase type, is as follows:

<table>
<thead>
<tr>
<th></th>
<th>TS</th>
<th>BIT</th>
<th>COMB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>( x^iyy )</td>
<td>2</td>
<td>( \frac{3}{2} )</td>
</tr>
<tr>
<td>(b)</td>
<td>( x^i(yx)^kyy )</td>
<td>( 2k )</td>
<td>( \frac{3}{2}k + 1 )</td>
</tr>
<tr>
<td>(c)</td>
<td>( x^i(yx)^kx )</td>
<td>( 2k - 1 )</td>
<td>( \frac{3}{2}k + 1 )</td>
</tr>
</tbody>
</table>

We provide a lower bound for \( OPT \) for each of these phases, thus obtaining the desired competitive ratios, for the three algorithms. To do this, we assign values to the dual variables, and obtain the lower bound from the dual objective function. In the next subsection we describe the assignment precisely.

### 2.4.2 The Dual Assignment

At time \( t = 1 \), we assign:

\[
s_{x,y} = b_{x,y,1} = x_{x,y,0},
\]
\[ s_{y,x} = b_{y,x,1} = x_{y,x,0}, \]

where \( x_{y,x,0}, x_{x,y,0} \) are the primal variables, based on the initial list state. That is, \( x_{x,y,0} = 1 \) if and only if item \( x \) precedes item \( y \) in the list at time \( t = 0 \), and \( x_{y,x,0} = 1 \) if and only if item \( y \) precedes item \( x \) in the list at time \( t = 0 \), i.e., \( x_{x,y,0} + x_{y,x,0} = 1 \).

Afterwards, the assignment is according to the phase type. We demonstrate the assignment on phases that begin when \( x \) is ahead of \( y \). If the order between \( x \) and \( y \) is reversed - the assignment is symmetric. All these phases begin with \( x^i \). For any time \( t \) matching the requests of \( x^i \) assign:

\[ a_{x,y,t} = a_{y,x,t} = 0, b_{x,y,t+1} = 1, b_{y,x,t+1} = 0. \]

We now show the assignment for every phase type separately:

(a) \( x^iyy \): For the first request to \( y \), assign:

\[ a_{x,y,t} = 1, b_{y,x,t+1} = 1, a_{y,x,t} = 0, b_{x,y,t+1} = 1. \]

For the second request to \( y \), assign:

\[ a_{x,y,t} = 0, b_{y,x,t+1} = 1, a_{y,x,t} = 0, b_{x,y,t+1} = 0. \]

(b) \( x^i(yx)^kyy \): For the last \( yy \), the assignment will be identical to the assignment in phase (a).

For any access to \( y \) during \((yx)^k\), assign:

\[ a_{x,y,t} = 1, b_{y,x,t+1} = 1, a_{y,x,t} = 0, b_{x,y,t+1} = 1. \]

For any access to \( x \) during \((yx)^k\) assign:

\[ a_{x,y,t} = 0, b_{y,x,t+1} = 0, a_{y,x,t} = 0, b_{x,y,t+1} = 1. \]

(c) \( x^i(yx)^kx \): For \((yx)^k\) the assignment is identical to the assignment in phase (b).

For the final \( x \), assign:

\[ a_{x,y,t} = 0, b_{y,x,t+1} = 0, a_{y,x,t} = 0, b_{x,y,t+1} = 1. \]

The only exception would be for time \( t = T \), for which we simply assign \( a_{x,y,T} = a_{y,x,T} = -\frac{1}{2} \), and there are no \( b \)-variables for time \( T + 1 \) to assign.

### 2.4.3 Proving Dual Feasibility

We first prove an invariant about the assignment, which will be useful for proving feasibility.
Lemma 2.4.2. For every phase beginning at time $t$, if item $x$ precedes item $y$ in the list at the beginning of time $t$ - then $b_{y,x,t} = 0$.

Proof. At time $t = 0$, we assigned $b_{y,x,1} = x_{y,x,0}$. So if item $x$ was ahead of $y$, then indeed $b_{y,x,1} = x_{y,x,0} = 0$.

If the previous phase was of type (a) or (b) - then items $x, y$ have switched places. We are now interested in the case where $x$ is ahead of $y$ at the end of the phase, that matches phases beginning when $y$ was ahead of $x$. Phase (a) would then be of the form: $y^i xx$. Phases of types (a) and (b), beginning with $y^i$, will both end with the same assignment. Say they ended at time $t' = t - 1$. In the previous subsection we described for phases (a),(b) starting with $x^i$, that the assignment will end with $b_{x,y,t'+1} = 0$. Since now our phase started with $y^i$, the assignment will end with $b_{y,x,t'+1} = 0 = b_{y,x,t}$, as we wanted to show. For phases of type (c) that start with $x^i$ - they end with two requests for item $x$, and not $y$, so for type (c) we will be interested in phases starting with $x$, that is, phases of the form (c) $x^i(yx)^kx$. In that case, our assignment also ends with $b_{y,x,t'+1} = 0 = b_{y,x,t}$, as we wanted to show.

We will simply go over the different constraints one by one, and show they are all satisfied.

The first set of constraints is:

$$\forall i \neq j, \forall t : 1 \leq t \leq T : b_{i,j,t} \leq 1.$$  

No $b$-variable ever gets higher than 1, so these constraints clearly hold.

The main constraints are:

$$\forall i \neq j, \forall t : 1 \leq t \leq T - 1, j = \sigma_t : a_{i,j,t} + a_{j,i,t} + b_{i,j,t} - b_{i,j,t+1} \leq 1,$$

and:

$$\forall i \neq j, \forall t : 1 \leq t \leq T - 1, j \neq \sigma_t : a_{i,j,t} + a_{j,i,t} + b_{i,j,t} - b_{i,j,t+1} \leq 0.$$

For times $t$ that match $x^i$ we assigned $a_{x,y,t} = a_{y,x,t} = 0, b_{x,y,t+1} = 1, b_{y,x,t+1} = 0$. From the lemma, we know that $b_{y,x,t} = 0$.

Since $b_{x,y,t} \leq 1$, we get:

$$a_{x,y,t} + a_{y,x,t} + b_{x,y,t} - b_{x,y,t+1} \leq 0 + 0 + 1 - 1 = 0,$$

$$a_{y,x,t} + a_{x,y,t} + b_{y,x,t} - b_{y,x,t+1} = 0 + 0 + 0 - 0 = 0,$$

and the constraints hold.

For phases of type (a), $x^iyy$, we assigned for the first request to $y$:

$$a_{x,y,t} = 1, b_{y,x,t+1} = 1, a_{y,x,t} = 0, b_{x,y,t+1} = 1.$$
The request is for $y$, so $\sigma_t = y$.

For $i = x, j = y = \sigma_t$, since $b_{x,y,t} \leq 1$, we get:

$$a_{x,y,t} + a_{y,x,t} + b_{x,y,t} - b_{x,y,t+1} \leq 1 + 0 + 1 - 1 = 1,$$

so the constraint holds.

From our lemma, we know that in the beginning of the phase $b_{y,x,t} = 0$, and as long as we are in the first $x^t$ our assignment was $b_{y,x,t+1} = b_{y,x,t} = 0$, so we know that $b_{y,x,t} = 0$.

Hence, for $i = y, j = x \neq \sigma_t$ we get:

$$a_{y,x,t} + a_{x,y,t} + b_{y,x,t} - b_{y,x,t+1} = 0 + 1 + 0 - 1 = 0,$$

so this constraint holds as well.

On the second request to item $y$, we assigned: $a_{x,y,t} = 0, b_{y,x,t+1} = 1, a_{y,x,t} = 0, b_{x,y,t+1} = 0$. The request is for $y$, so $\sigma_t = y$.

For $i = x, j = y = \sigma_t$, since $b_{x,y,t} \leq 1$, we get:

$$a_{x,y,t} + a_{y,x,t} + b_{x,y,t} - b_{x,y,t+1} \leq 0 + 0 + 1 - 1 = 1,$$

so the constraint holds.

For $i = y, j = x \neq \sigma_t$, since $b_{y,x,t} \leq 1$, we get:

$$a_{y,x,t} + a_{x,y,t} + b_{y,x,t} - b_{y,x,t+1} \leq 0 + 0 + 1 - 1 = 0,$$

so this constraint holds as well.

Phases of type (b) also end with $yy$. For this part, since the assignment is identical to the previous phase, the analysis is also identical to what was shown for (a). We only need to show that $b_{y,x,t} = 0$ during the first of the two requests for $y$. This is true, since the previous request was for $x$, for which we have indeed assigned $b_{y,x,t} = 0$ (defined in the previous subsection), as required.

For the requests $(yx)^k$, during a request for $y$ we assigned:

$$a_{x,y,t} = 1, b_{y,x,t+1} = 1, a_{y,x,t} = 0, b_{x,y,t+1} = 1.$$

The request is for $y$, so $\sigma_t = y$.

For $i = x, j = y = \sigma_t$, since $b_{x,y,t} \leq 1$, we get:

$$a_{x,y,t} + a_{y,x,t} + b_{x,y,t} - b_{x,y,t+1} \leq 1 + 0 + 1 - 1 = 1,$$

so the constraint holds.

For $i = y, j = x \neq \sigma_t$ we get:

$$a_{y,x,t} + a_{x,y,t} + b_{y,x,t} - b_{y,x,t+1} \leq 0 + 0 + 1 - 1 = 0,$$
so this constraint holds as well.

During the request for $x$ we assigned $a_{x,y,t} = 0, b_{y,x,t+1} = 0, a_{y,x,t} = 0, b_{x,y,t+1} = 1$. The request is for $x$, so $\sigma_t = x$.

For $i = x, j = y \neq \sigma_t$, since $b_{x,y,t} \leq 1$, we get:

$$a_{x,y,t} + a_{y,x,t} + b_{x,y,t} - b_{x,y,t+1} \leq 0 + 0 + 1 - 1 = 0,$$

so the constraint holds.

For $i = y, j = x = \sigma_t$, since $b_{y,x,t} \leq 1$, we get:

$$a_{y,x,t} + a_{x,y,t} + b_{y,x,t} - b_{y,x,t+1} \leq 0 + 0 + 1 - 0 = 1,$$

so this constraint holds as well.

For phases of type (c), they start out with the same assignment and analysis as type (b). They end with a request for $x$, at which we assigned:

$$a_{x,y,t} = 0, b_{y,x,t+1} = 0, a_{y,x,t} = 0, b_{x,y,t+1} = 1.$$

The request is for $x$, so $\sigma_t = x$.

For $i = x, j = y \neq \sigma_t$, since $b_{x,y,t} \leq 1$, we get:

$$a_{x,y,t} + a_{y,x,t} + b_{x,y,t} - b_{x,y,t+1} \leq 0 + 0 + 1 - 1 = 0,$$

so the constraint holds.

For $i = y, j = x = \sigma_t$, since $b_{y,x,t} \leq 1$, we get:

$$a_{y,x,t} + a_{x,y,t} + b_{y,x,t} - b_{y,x,t+1} \leq 0 + 0 + 1 - 0 = 1,$$

so this constraint holds as well.

The next set of constraints is for time $T$:

$$\forall i \neq j, j = \sigma_T : a_{i,j,T} + a_{j,i,T} + b_{i,j,T} \leq 1,$$

$$\forall i \neq j, j \neq \sigma_T : a_{i,j,T} + a_{j,i,T} + b_{i,j,T} \leq 0.$$

We assigned $a_{x,y,T} = a_{y,x,T} = -\frac{1}{2}$, so these constraints clearly hold.

The last set of constraints is for time $t = 0$:

$$\forall i \neq j : s_{i,j} - b_{i,j,1} \leq 0.$$

But we have defined $s_{i,j} - b_{i,j,1} = x_{i,j,0} - x_{i,j,0} = 0$, so this constraint clearly holds as well.
2.4.4 Value of Dual Solution

The dual objective function is:

\[
\sum_{i=1}^{l} \sum_{j=1}^{l} \sum_{t=1}^{T} a_{i,j,t} + \sum_{i=1}^{l} \sum_{j=i+1}^{l} s_{i,j}.
\]

There are only \(2\binom{l}{2}\) variables for \(s_{i,j}\), and \(2\binom{l}{2}\) variables for \(a_{i,j,t}\), and since their values are all greater then \((-1)\), so their sum can be lower bounded by an additive constant of \(C = -4\binom{l}{2}\). The significant part of the value of the dual solution - which will determine the multiplicative factor - comes from the assignments for the three phases, as described.

For phases of type (a) \(x'y'\), we have assigned \(a_{x,y,t} = 1\) once for the first request to \(y\), and \(a_{i,j,t} = 0\) for the rest. So, for phases of type (a), the dual solution value is 1 for every phase of type (a).

For phases of type (b) \(x'(yx)^k'y'\), for every \((yx)\) we have assigned a single \(a\)-variable with the value 1, and the rest with 0. That gives us a value of \(k\) units for \((yx)^k\). For the final \(yy\), our assignment was as in phase (a), getting one additional unit, and thus a total of \(k + 1\), for every phase of type (b).

For phases of type (c) \(x'(yx)^k'x\), for the \((yx)^k\) we assign the same values as in phases of type (b) thus gaining the same \(k\) units. For the final request to \(x\), we assigned \(a_{x,y,t} = a_{y,x,t} = 0\), which leaves us with a total of \(k\) units for every phase of type (c).

2.4.5 Obtaining the Competitiveness of TS, BIT, And COMB

We now summarize the costs of all algorithms, and the value of the dual solution, for each phase type.

This is as follows:

<table>
<thead>
<tr>
<th></th>
<th>TS</th>
<th>BIT</th>
<th>COMB</th>
<th>DUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(x'y')</td>
<td>(2)</td>
<td>(\frac{3}{2})</td>
<td>(\frac{8}{5})</td>
</tr>
<tr>
<td>(b)</td>
<td>(x'(yx)^k'y')</td>
<td>(2k)</td>
<td>(\frac{3}{2}k + 1)</td>
<td>(\frac{3}{5}k + \frac{4}{5})</td>
</tr>
<tr>
<td>(c)</td>
<td>(x'(yx)^k'x)</td>
<td>(2k - 1)</td>
<td>(\frac{3}{2}k + \frac{1}{4})</td>
<td>(\frac{3}{5}k)</td>
</tr>
</tbody>
</table>

For the algorithm \(TS\), we can see that for phases of type (a) \(\frac{TS(a)}{DUAL(a)} = \frac{2}{1} = 2\).

For phases of type (b) \(\frac{TS(b,k)}{DUAL(b,k)} = \frac{2k}{k+1} \leq 2\).

For phases of type (c) \(\frac{TS(c,k)}{DUAL(c,k)} = \frac{2k-1}{k} \leq 2\).

From neglecting the end of a phase, we might get an additional cost of at most \(c = 2\binom{l}{2}\) for the cost of \(TS\).

We use \#a, \#b, \#c to denote the number of phases of types (a), (b), and (c) in the input sequence \(\sigma\), respectively.
We use $k_{b,i}, k_{c,i}$ to denote the number $k$ of repetitions of $(yx)$ (or $(xy)$), in the $i$-th phase of type (b),(c) in $\sigma$, respectively.

$$DUAL(\sigma) \geq \#aDUAL(a) + \sum_{i=1}^{\#b} DUAL(b, k_{b,i}) + \sum_{i=1}^{\#c} TS(c, k_{c,i}) + C.$$ 

$C = -4\left(\frac{1}{2}\right)$ is the constant we may have lost for the dual solution from the variables $s_{i,j}$, and $a_{i,j,T}$.

Thus, in total:

$$TS(\sigma) \leq \#aTS(a) + \sum_{i=1}^{\#b} TS(b, k_{b,i}) + \sum_{i=1}^{\#c} TS(c, k_{c,i}) + c \leq$$ 

$$2\#aDUAL(a) + 2\sum_{i=1}^{\#b} DUAL(b, k_{b,i}) + 2\sum_{i=1}^{\#c} DUAL(c, k_{c,i}) + c \leq$$ 

$$2DUAL(\sigma) - C + c \leq 2OPT(\sigma) - C + c.$$ 

And hence $TS$ is a 2-competitive algorithm.

For the algorithm $BIT$, we can see that for phases of type (a) $\frac{BIT(a)}{DUAL(a)} = \frac{8}{5} \leq 1.75$.

For phases of type (b) $\frac{BIT(b,k)}{DUAL(b,k)} = \frac{2k+1}{k+1} \leq 1.75$.

For phases of type (c) $\frac{BIT(c,k)}{DUAL(c,k)} = \frac{2k+2}{2k} \leq 1.75$, since $k \geq 1$.

Thus, in total:

$$BIT(\sigma) \leq \#aBIT(a) + \sum_{i=1}^{\#b} BIT(b, k_{b,i}) + \sum_{i=1}^{\#c} cBIT(c, k_{c,i}) + c \leq$$ 

$$1.75\#aDUAL(a) + 1.75\sum_{i=1}^{\#b} DUAL(b, k_{b,i}) + 1.75\sum_{i=1}^{\#c} DUAL(c, k_{c,i}) + c \leq$$ 

$$1.75DUAL(\sigma) - C + c \leq 1.75OPT(\sigma) - C + c.$$ 

And hence $BIT$ is a 1.75-competitive algorithm.

For the algorithm $COMB$, we can see that for phases of type (a) $\frac{COMB(a)}{DUAL(a)} = \frac{8}{5} \leq 1.6$.

For phases of type (b) $\frac{COMB(b,k)}{DUAL(b,k)} = \frac{2k+4}{k+1} \leq 1.6$.

For phases of type (c) $\frac{COMB(c,k)}{DUAL(c,k)} = \frac{2k}{k} \leq 1.6$.

Thus, in total:

$$COMB(\sigma) \leq \#aCOMB(a) + \sum_{i=1}^{\#b} COMB(b, k_{b,i}) + \sum_{i=1}^{\#c} COMB(c, k_{c,i}) + c \leq$$ 

$$21.$$
$$1.6\#aDUAL(a) + 1.6 \sum_{i=1}^{\#b} DUAL(b, k_{b,i}) + 1.6 \sum_{i=1}^{\#c} DUAL(c, k_{c,i}) + c \leq$$

$$1.6DUAL(\sigma) - C + c \leq 1.6OPT(\sigma) - C + c.$$  

And hence \textit{COMB} is a 1.6-competitive algorithm.
Chapter 3

Lower Bound For Free Exchanges Only

In this chapter, we consider the offline list update problem. In the offline scenario, the entire request sequence is given to us in advance, and we may plan our actions accordingly. As shown in Lemma 2.1.1, using only paid exchanges still allows us to solve the list update problem optimally. However, what about the converse? Can we solve the problem to optimality using only free exchanges? In [LORR15], a lower bound of $\frac{12}{11}$ was shown on the ratio between the optimum using free exchanges only, and the general optimum, that may use both free and paid exchanges. In this chapter, we improve this lower bound, and show a lower bound of $1.25$ on this ratio. The result is significant in teaching us about the limitations of the use of free exchanges only. Using only free exchanges we cannot get a better approximation factor than $1.25$. Note, that the algorithms considered so far - MTF, BIT, TS, and COMB - use only free exchanges. The result shows that to get an approximation better than $1.25$ we would have to take a different approach than the algorithms above, and use paid exchanges as well.

To prove the desired lower bound, we present an example of a list and a sequence of requests, for which we show that the ratio between the optimum using free exchanges only is $1.25$ times the general optimum. In fact, our example uses a parameter $k$. We calculate the costs of both optima as a function of $k$, and show that as $k$ approaches infinity, the ratio approaches $1.25$. We also prove that the two presented optima are indeed optimal.

3.1 Defining the Example

We use $OPT$ to denote an unrestricted optimal offline algorithm, and we use $OPT - FREE$ to denote an optimal offline algorithm restricted to using only free exchanges. For the request sequences, we denote multiple requests in a row to the same item by using exponents, e.g. $x^k$ means that item $x$ is requested $k$ times in a row.
We look at a list \( L \) with \( 2k \) items, denoted as \( x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{2k} \). For the list \( L \) we define two specific sequences of requests as follows: \( \sigma_1 = < x_{2k}, x_{2k-1}, \ldots, x_{k+1} > \), \( \sigma_2 = < x_{3k+1}^3, x_{3k+2}^3, \ldots, x_{2k}^3 > \). We define \( R(L) = \sigma_1 \sigma_2 = \sigma \). We show that by taking \( k \) to infinity we get a multiplicative gap of 1.25 between the optimum using only free exchanges, and the optimum using both free and paid exchanges. After serving this sequence using the optimal algorithm, which we denote by \( OPT \), the list ends in a certain order. We relabel the items according to that order, so that the item that is currently first is now relabelled as \( x_1 \), the item that is currently second - is now relabelled as \( x_2 \), and so on. After relabelling, we repeat the sequence \( \sigma \) again. And then we do another relabelling, and another repetition of \( \sigma \), and so on. Our complete input sequence is composed of the concatenation of the input from all the repetitions. The number of repetitions can be arbitrarily large, which is a must for arguing that the gap is indeed multiplicative, and not additive.

## 3.2 The Cost Of OPT

Intuitively, when accessing the items in \( \sigma_1 \), we wish to move them ahead of \( x_1, \ldots, x_k \) without changing their internal order, so that they will be in place for \( \sigma_2 \). That can only be done using paid exchanges. First, we calculate the cost \( OPT(\sigma) \). Second, we prove that \( OPT_FREE \) must move any item to the head of the list upon three consecutive requests, in order to retain optimality. Third, we show that \( OPT_FREE \) must move any item to the head of the list upon any request in our input sequence, and that for our specific input sequence \( OPT_FREE \) in fact behaves identically to the well-known MTF algorithm [ST85]. Finally, we show the ratio between \( OPT_FREE \) and \( OPT \) for our input sequence.

### 3.2.1 Calculating the Cost of OPT

Let us compute the cost of an optimal algorithm \( OPT \), that uses paid exchanges. \( OPT \) first moves the first \( k \) items to the end of the list, incurring a cost of \( k \) per item, and a total cost of \( k^2 \). Then, servicing \( \sigma_1 \) would cost \( \sum_{i=1}^{k} i = \frac{k(k+1)}{2} \). Then, \( OPT(\sigma_2) = \sum_{i=1}^{k} i + 2k = \frac{k(k+1)}{2} + 2k \). The first term accounts for the first access to each item, which is then moved to the front of the list. The second term accounts for the two remaining accesses, with a cost of one each, for a total of \( k \) items. So, in total, we get \( OPT(\sigma) = k^2 + k(k+1) + 2k = 2k^2 + 3k \).

We wish to find a lower bound for the ratio between \( OPT_FREE \) and \( OPT \). For \( OPT_FREE \), we need a lower bound, and so we must prove that it is indeed optimal. However, for \( OPT \), we only need an upper bound, and the algorithm we described above can serve as an upper bound for the cost of \( OPT \), without having to prove it is, indeed, optimal. If there exists an algorithm with a lower cost, then the lower bound for the ratio between \( OPT_FREE \) and \( OPT \) would be even larger. However, for the
completeness of our work, we now present a proof that \textit{OPT} is indeed an optimal algorithm for the described input sequence.

3.2.2 \textbf{OPT is Optimal}

To prove this, we rely on the following two theorems. The first was shown by Reingold and Westbrook [RW96]:

\textbf{Theorem 3.1.} If an item \(x\) is requested 3 or more times consecutively, then an optimal algorithm must move it to the front before the second access.

The second theorem was show by Borodin and El-Yanin [BEY98]. Is states that in the partial cost model, the partial cost of an algorithm is the sum of partial costs for all pairs.

\textbf{Theorem 3.2.} Under the partial cost model, for any algorithm \(\text{alg}\), for any input sequence \(\sigma\), the following equality holds:

\[ \text{alg}(\sigma) = \sum_{x \neq y} \text{alg}(\sigma_{xy}). \]

The notation \(\sigma_{xy}\) is used for the projection of \(\sigma\) on the items \(x, y\), which is done by removing requests for all other items. For instance, if \(\sigma = \langle x, y, z, x, z, y \rangle\), then \(\sigma_{xy} = \langle xyxy \rangle\).

We denote the number of repetitions of \(\sigma\) by \(4a_4 + b_4\), where \(a_4\) is a non-negative integer, and \(b_4 \in \{0, 1, 2, 3\}\). Alternatively, the number of repetitions is \(2a_2 + b_2\), where \(a_2\) is a non-negative integer, and \(b_2 \in \{0, 1\}\). We denote the complete input sequence by \(\sigma^*\). We now show that \textit{OPT} is an optimal algorithm for the defined input sequence.

\textbf{Theorem 3.3.} \textit{OPT} is an optimal algorithm for \(\sigma^*\).

\textit{Proof.} We now remove the relabelling and examine the sequence as is. We have defined the sequence to be composed of \(R(L) = \sigma = \langle x_{2k}, \ldots, x_{k+1}, x_k^3, \ldots, x_{2k}^3 \rangle\) when \textit{OPT} begins with a list order of \(L = \langle x_1, x_2, \ldots, x_{2k} \rangle\). In serving one iteration of \(\sigma = \langle x_{2k}, \ldots, x_{k+1}, x_k^3, \ldots, x_{2k}^3 \rangle\), \textit{OPT} moves the items \(x_1, \ldots, x_k\) to the back of the list using paid exchanges. Then, each item that is accessed three times consecutively, is moved to the front of the list. So after the first iteration, the items \(x_{k+1}, \ldots, x_{2k}\) are at the front with reversed order, and the items \(x_1, \ldots, x_k\) are at the back, in their original order. So the list order after serving \(\sigma\) is: \(L = \langle x_{2k}, \ldots, x_{k+1}, x_1, \ldots, x_k \rangle\). And then, for the second iteration we get \(R(L) = \langle x_k, \ldots, x_1, x_{2k}^3, \ldots, x_k^3 \rangle\). During the second iteration, items \(x_1, \ldots, x_k\) move to the head of the list, reversing their order, and items \(x_{k+1}, \ldots, x_{2k}\) move to the back, maintaining their order. Thus, the list order after the second iteration is \(L = \langle x_1, x_{2k}, \ldots, x_{k+1} \rangle\). And now for the third iteration we get \(R(L) = \langle x_{k+1}, \ldots, x_{2k}, x_k^3, \ldots, x_{k+1}^3 \rangle\). During the third
iteration, items \( x_{k+1}, \ldots, x_{2k} \) move to the head of the list, reversing their order, and items \( x_1, \ldots, x_k \) move to the back, maintaining their order. Thus, the list order after the third iteration is: \( L = \langle x_{k+1}, \ldots, x_{2k}, x_k, \ldots, x_1 \rangle \). And now for the fourth iteration we get \( R(L) = \langle x_1, \ldots, x_k, x_{k+1}^3, \ldots, x_1^3 \rangle \). During the fourth iteration, items \( x_1, \ldots, x_k \) move to the head of the list, reversing their order, and items \( x_{k+1}, \ldots, x_{2k} \) move to the back, maintaining their order. Thus, the list order after the fourth iteration is: \( L = \langle x_1, \ldots, x_k, x_{k+1}, \ldots, x_{2k} \rangle \). So after four iterations of \( \sigma \) we have returned to the original list order. Let us simply write down the four request sequences \( R(L) \) one after the other, together. The total input sequence looks as follows:

\[
\sigma' = \langle x_{2k}, \ldots, x_{k+1}, x_k^3, \ldots, x_{2k}^3, x_k^3, \ldots, x_1^3, x_k^3, \ldots, x_1^3 \rangle
\]

\( \sigma' \) is composed of four repetitions of \( \sigma \) without relabelling. After four repetitions served by \( \text{OPT} \), the list returns to its original order, and so the next repetitions will be just as these four ones.

We denote the cost of an algorithm \( \text{alg} \) over the sequence \( \sigma \) by \( \text{alg}(\sigma) \).

For analyzing \( \text{OPT} \), we consider the partial cost of it. We analyze the costs of pairs of items, and show that \( \text{OPT} \) is optimal for each pair separately. From 3.2, we conclude that \( \text{OPT} \) is an optimal algorithm.

We prove our theorem using the partial cost model for simplicity, because an algorithm is optimal in the partial cost model if and only if it is optimal in the full cost model (since the difference between the two cost models is simply the input length, and is a constant independent of the algorithm). We have calculated the cost of the algorithm in the full cost model, for the sake of calculating the gap between algorithms that may use paid exchanges, and algorithms that may not.

We now consider the partial cost for each pair \( x_i, x_j \) separately. We assume w.l.o.g. \( i < j \). There are three types of pairs \( x_i, x_j \), as follows:

1. \( i \leq k < j \) - One item is from the lower indices and one from the higher indices.
2. \( k < i < j \) - Both items are from the higher indices.
3. \( i < j \leq k \) - Both items are from the lower indices.

For pairs of type (1), the projection of the input sequence over the items \( x_i, x_j \) is \( \sigma*_{ij} = (x_j^4 x_i^4)^{a_2}(x_j^4)^{b_2} \). \( \text{OPT} \) pays one unit cost for every repetition of \( \sigma \), when it swaps the two items using a paid exchange. Afterwards, all accesses are done when the accessed item is the first one of the two, so it costs no more. From Theorem 3.1, we know that \( \text{OPT} \) must swap between \( x_i, x_j \) at every repetition of \( \sigma \), and thus - it must pay at least one unit cost for every repetition, and thus \( \text{OPT} \) is optimal for pairs of type (1).

For pairs of type (2), the projection of the sequence \( \sigma* \) over the items \( x_i, x_j \) starts with \( (x_j x_i x_j^3 x_i x_j^3)^{a_4} \). If \( b_4 = 0 \) - that is the entire sequence. If \( b_4 \in 1, 2 \) it ends with
an additional $x_jx_i^4x_j^3$, and if $b_4 = 3$ it ends with an additional $x_jx_i^4x_j^3x_i^4x_j^3$. OPT always puts an item first before 3 or 4 consecutive requests for that item, as an optimal algorithm must do. OPT does not swap between them for single requests. Swapping costs one unit for the swap itself, saves one unit for the serving the single request, but then leaves the accessed item ahead of the other one, which is requested next. Thus, it costs another unit for the next request, and is not optimal. For this reason, OPT does not swap between the items for a single request, and is thus optimal for any pair of type (2).

For pairs of type (3), the analysis is identical to pairs of type (2).

Therefore, OPT achieves the optimal value for every pair of items $x_i, x_j$ separately, and is thus an optimal algorithm for $\sigma*$.

\[\square\]

### 3.3 The Cost Of OPT_FREE

To calculate the cost of OPT_FREE, we need to understand its behavior. This is done with the next two lemmas.

#### 3.3.1 Three Consecutive Requests

We show that there exists an optimal OPT_FREE that moves items to the head of the list during $\sigma_2$, and we choose OPT_FREE to behave in this manner.

**Lemma 3.3.1.** There exists an optimal OPT_FREE that moves every item to the head of the list upon its request.

**Proof.** We first present an outline of the proof, and then provide the formal details.

We consider an algorithm ALG, that uses only free exchanges, and does not move all items to the front upon three consecutive requests. We replace ALG with an algorithm ALG', that moves more items to the front upon three consecutive requests, and performs at least as well as ALG. This procedure can be repeated, until we get an algorithm that moves all items to the front of the list upon three consecutive requests, and performs just as well. This means we can do the same process with any optimal OPT_FREE, and get ourselves a version that is still optimal, and always moves items to the front upon three consecutive requests.

For analyzing ALG' we consider its partial cost. We analyze the costs of pairs of items. Similarly to proving the optimality of OPT, we consider the same three different types of pairs, marked as (1),(2),(3). We show that:

For pairs of type (1) ALG' is optimal.

For pairs of types (2),(3) - ALG' costs one unit more than the optimum. For every such pair in which ALG is optimal, there is a distinct pair of type (1) for which ALG is not optimal (and ALG' is), which means that in total ALG is not better then ALG'.

We now proceed to the formal proof.
We show that we can replace any algorithm $ALG$ that does not move all items to the front of the list upon three consecutive requests, i.e. during $\sigma_2$, with an algorithm $ALG'$ that "moves more items" to the front of the list upon three consecutive requests, without increasing its cost, i.e. having $ALG(\sigma) \geq ALG'(\sigma)$. Note that "Moving more items" actually means we replace $ALG$ with an algorithm $ALG'$ that moves all items to the front upon three consecutive requests, from a particular point in the input and onwards, which is earlier then the one from which $ALG$ moved all items to the front upon three consecutive requests. After showing that, we may repeatedly replace $ALG$ with $ALG'$ until that point of the input is the beginning of the input, and then we are done.

Let us consider an algorithm $ALG$ that does not always move items to the front of the list upon three consecutive requests, i.e. during $\sigma_2$. Let us look at the last instance of $\sigma_2$ where $ALG$ does not move all items to the front of the list. The input from that point on is $\sigma_2 \sigma_1 \sigma_2 \sigma_1 \sigma_2 \sigma_1 \sigma_2 \ldots$. We denote the current $\sigma_2$ by $A$, the next $\sigma_1$ by $B$, and afterwards $C, D, E$ etc. If $A$ or $C$ are the last parts of the input, then we use $MTF$ from the beginning of $A$ and until the end of the input (in $A$ or $C$). We analyze this case later separately, but for now let us assume it is not the case, i.e. the input ends at $E$ or later. We assume wlog (otherwise we can just relabel) that $A = \langle x_{k+1}^3, x_{k+2}^3, \ldots, x_{2k}^3 \rangle, B = \langle x_k, \ldots, x_1 \rangle, C = \langle x_1^3, \ldots, x_k^3 \rangle, D = \langle x_{k+1}, \ldots, x_{2k} \rangle, E = \langle x_{2k}^3, \ldots, x_{k+1}^3 \rangle$. We replace $ALG$ with a different algorithm $ALG'$. Up until the beginning of $A$, $ALG'$ behaves exactly as $ALG$. For the segments $A, B, C, D, E$, $ALG'$ behaves like $MTF$. Since we picked $A$ to be the last $\sigma_2$ where $ALG$ does not move all items to the front, we know that in $C$, $ALG$ does move the items $x_1, \ldots, x_k$ to the front, and in $E$ $ALG$ does move the items $x_{k+1}, \ldots, x_{2k}$ to the front, just like $MTF$. Consequently, after $E$ the list of $ALG'$ returned to the same state as $ALG$. Namely, the order of items in both lists is: $L = x_{k+1}, x_{k+2}, \ldots, x_{2k}, x_k, x_{k-1}, \ldots, x_1$. Therefore, after $E$ is finished, the items manipulated by $ALG'$ are in the same order as in $ALG$ at that point. So, after serving $E$ we can and will define $ALG'$ to behave like $ALG$. So the only differences between the behaviour of the two algorithms are during $ABCDE$.

We now denote $\sigma* = ABCDE$.

We prove our lemma using the partial cost model for simplicity, because an algorithm is optimal in the partial cost model if and only if it is optimal in the full cost model (since the difference between the two cost models is simply the input length, and is a constant independent of the algorithm). We later calculate the cost of the algorithm in the full cost model, for the sake of calculating the gap between algorithms that may use paid exchanges, and algorithms that may not.

We now consider the partial cost for each pair $x_i, x_j$ separately. We assume wlg $i < j$. There are three types of pairs $x_i, x_j$, as follows:

1. $i \leq k < j$ - One item is from the lower indices and one from the higher indices.
2. $k < i < j$ - Both items are from the higher indices.

28
(3) \( i < j \leq k \) - Both items are from the lower indices.

For pairs of type (1), the projection of the sequence \( \sigma^* \) over the items \( x_i, x_j \) is \( \sigma^*_{ij} = x_i^j x_j^i x_i^j \). For this sequence, \( MTF \) is clearly optimal, so \( ALG' \) can only improve compared to \( ALG \). \( ALG' \)'s optimality means: \( ALG'(\sigma^*_{ij}) = MTF(\sigma^*_{ij}) = OPT(\sigma^*_{ij}) \leq ALG(\sigma^*_{ij}). \)

For pairs of type (2), the projection of the sequence \( \sigma^* \) over the items \( x_i, x_j \) is \( \sigma^*_{ij} = x_i^j x_j^i x_i^j x_i^j. \) For this sequence, \( MTF \) costs one unit more than the optimum, which would leave \( x_i \) in place during the single access to it. That is, \( ALG'(\sigma^*_{ij}) = MTF(\sigma^*_{ij}) = OPT(\sigma^*_{ij}) + 1. \) If the original is not optimal: \( ALG(\sigma^*_{ij}) \geq OPT(\sigma^*_{ij}) + 1 \) - then \( ALG'(\sigma^*_{ij}) \leq ALG(\sigma^*_{ij}) \) and we are good. Otherwise - \( ALG(\sigma^*_{ij}) = OPT(\sigma^*_{ij}) = ALG'(\sigma^*_{ij}) - 1. \) In this case, we notice that the single access to \( x_i \) in \( \sigma^*_{ij} \) is during part \( D \) of \( \sigma^* \). In part \( C \), \( ALG \) moved the items \( x_1, ..., x_k \) to the front, so if \( ALG \) does not move \( x_j \) ahead of \( x_j \), it cannot move \( x_i \) ahead of \( x_i, ..., x_k \) either. We denote by \( x_f \) an item from \( x_1, ..., x_k \). This means, that the performance of \( ALG \) for the \( k \) pairs \( (x_1, x_1), ..., (x_k, x_i) \) costs at least one unit more than the optimum, which is obtained by \( ALG' \), for each such pair \( (x_f, x_i) \). That is, \( ALG'(\sigma^*_{fj}) = MTF(\sigma^*_{fj}) = OPT(\sigma^*_{fj}) \), whereas \( ALG'(\sigma^*_{fj}) + 1 = OPT(\sigma^*_{fj}) + 1 \leq ALG(\sigma^*_{fj}) \). For every \( x_i \) there are \( k \) such pairs (of type (1)), and at most \( k - 1 \) pairs of type (2) (since \( i \geq k + 1 \) there are at most \( k - 1 \) items \( x_j \) having \( j > i \)). Hence, in total, \( ALG \) does not outperform \( ALG' \), and only loses for every \( x_i \) it chooses to leave in place during \( D \).

For pairs of type (3), the projection of the sequence \( \sigma^* \) over the items \( x_i, x_j \) is \( \sigma^*_{ij} = x_i^j x_j^i x_i^j \). If \( x_j \) starts ahead of \( x_i \) (at the beginning of \( A \), where the lists of \( ALG \) and \( ALG' \) are identical) - then \( MTF \) is optimal for this pair, and we are good. Otherwise, \( MTF \) costs one unit more than the optimum, which would leave \( x_j \) after \( x_i \) during the first access to it. That is, \( ALG'(\sigma^*_{ij}) = MTF(\sigma^*_{ij}) = OPT(\sigma^*_{ij}) + 1. \) If the original is not optimal: \( ALG(\sigma^*_{ij}) \geq OPT(\sigma^*_{ij}) + 1 \) - then \( ALG'(\sigma^*_{ij}) \leq ALG(\sigma^*_{ij}) \) and we are good. Otherwise - \( ALG(\sigma^*_{ij}) = OPT(\sigma^*_{ij}) = ALG'(\sigma^*_{ij}) - 1. \) However, we notice that the first access to \( x_j \) is during part \( B \). Let us examine items of higher indices \( x_f \) s.t. \( f > k \). The two pairs \( (x_i, x_f), (x_j, x_f) \) are both of type (1), and therefore \( ALG' \) is optimal for both.

If \( ALG \) moved \( x_f \) ahead of \( x_i \) during segment \( A \), then in order for \( ALG \) to perform optimally for the pair \( x_i, x_j \), \( ALG \) needs to leave \( x_j \) after \( x_i \), which is after \( x_f \). This means that \( ALG' \) outperforms \( ALG \) for the pair \( (x_j, x_f) \) by one unit cost (at least). That is, \( ALG'(\sigma^*_{jj}) \leq ALG(\sigma^*_{ff}) - 1. \) Otherwise - \( ALG \) did not move \( x_f \) ahead of \( x_i \) during \( A \), which means \( ALG' \) outperformed \( ALG \) for the pair \( x_i, x_f \) by at least one unit cost. That is, \( ALG'(\sigma^*_{if}) \leq ALG(\sigma^*_{if}) - 1. \) So - for every pair \( (x_i, x_j) \) : either \( ALG'(\sigma^*_{jj}) \leq ALG(\sigma^*_{jj}) - 1 \) or \( ALG'(\sigma^*_{if}) \leq ALG(\sigma^*_{if}) - 1 \), for every \( f \in \{k + 1, ..., 2k\} \). Let us examine a specific \( x_f \). If for only \( n \) out of \( k \) values in \( \{1, 2, ..., k\} \) the following inequality is satisfied: \( ALG'(\sigma^*_{jj}) \leq ALG(\sigma^*_{jj}) - 1 \). These \( n \) values are called special values. Then - for pairs of \( i, j \) who are both part of the other \( n - k \), we know that \( ALG'(\sigma^*_{ij}) \leq ALG(\sigma^*_{ij}) \) (Since we just showed that if \( ALG \) is
optimal for \((x_i, x_j)\) - then either \(i\) or \(j\) are special). Thus, the number of pairs \(i, j\) left with \(ALG(\sigma*_{ij}) = OPT(\sigma*_{ij}) = ALG'(\sigma*_{ij}) - 1\) is at most \(p = \frac{n(n-1)}{2} + n(k - n)\). The first term from pairs \(i, j\) who are both from the special \(n\), and the second term from involved pairs (one special value, one not special). Calculating: \(p = \frac{n(n-1)}{2} + n(k - n) \leq n^2 + nk - n^2 = nk\). That is, if we have only \(n\) special values - then we have at most \(nk\) such pairs. Hence, if we have \(nk\) such pairs - we have at least \(n\) special values, for every specific \(x_f\). Thus, in total, at least \(nk\) special values counting over all values of \(f\).

Thus, if we have \(m\) pairs for which \(ALG'\) loses one unit cost, compared to \(ALG\) - then we also have at least \(m\) special values, for which \(ALG'(\sigma*_{ij}) \leq ALG(\sigma*_{ij}) - 1\). Hence, in total, \(ALG\) does not outperform \(ALG'\).

A potential problem with this proof would be if we have counted the same pair of type \((1)\) twice in our favor against two different pairs of types \((2)\) or \((3)\). For two pairs of type \((2)\) during our calculation we explicitly counted different pairs. Same is true for different pairs of type \((3)\). However - What about counting the same pair \((x_i, x_j)\) of type \((1)\), once to account for a pair of type \((2)\), and once for a pair of type \((3)\)? Actually - it is possible that we have done that, but in this case we will now show that \(ALG'(\sigma*_{ij}) \leq ALG(\sigma*_{ij}) - 2\), i.e. we gain at least two units cost for such pairs comparing to \(ALG\), so we may count them twice. We saw that \(\sigma*_{ij} = x_i^j x_i^j\). If we counted the pair \((x_i, x_j)\) for a pair of type \((2)\) - then \(ALG\) did not move \(x_j\) ahead of \(x_i\) during the first out of the four consecutive requests to it, which were during part \(D\). So \(ALG'((DE)_{ij}) \leq ALG((DE)_{ij}) - 1\). If we counted the pair \((x_i, x_j)\) for a pair of type \((3)\) - then either \(ALG\) did not move \(x_i\) ahead of \(x_j\) in the first out of four consecutive requests for it, during part \(B\). Or - \(ALG\) did not move \(x_j\) ahead of \(x_i\) in any out of the three consecutive requests for it, during part \(A\). In any case, \(ALG'((ABC)_{ij}) \leq ALG((ABC)_{ij}) - 1\). In total, \(ALG'(\sigma*_{ij}) \leq ALG(\sigma*_{ij}) - 2\). Thus, counting the pair twice was justified. Therefore, the total payment of the algorithms for all pairs satisfies: \(ALG'(\sigma*) \leq ALG(\sigma*)\). Finally, since for the rest of \(\sigma\) the two algorithms behaved identically, then \(ALG'(\sigma) \leq ALG(\sigma)\) - that is, \(ALG\) did not outperform \(ALG'\) in total.

Now for the cases we have neglected - if the input sequence ends at \(A\), or \(C\). If it ends at \(A\), then \(MTF\) is optimal for all 3 types of pairs in serving \(A\). Thus, \(ALG\) clearly does not outperform \(ALG'\).

If \(C\) is the end of the input sequence, \(ALG\) might be better then \(ALG'\) only for pairs of type \((3)\). But as described above, there are necessarily more pairs of type \((1)\) for which \(ALG'\) outperforms \(ALG\), and hence \(ALG'\) is better in total (same proof exactly for pairs of type \((3)\)).

Thus, we can replace any algorithm \(ALG\) that does not move all items to the front upon three consecutive requests, with an algorithm \(ALG'\) that moves more items to the front upon three consecutive requests, and performs at least as good. Hence, we can repeatedly replace an optimal algorithm by \(ALG'\), until we reach an algorithm that is both optimal, and guarantees to move each item to the front upon three consecutive
requests. Therefore, there exists an optimal algorithm that moves each item to the front upon three consecutive requests for this specific input sequence.

3.3.2 Understanding the Optimal Algorithm OPT_FREE

Now, let us examine the optimum with free exchanges only, \textit{OPT\_FREE}. Let us look closely at an item \(x_i\) having \(k + 1 \leq i \leq 2k\). This item is accessed first in \(\sigma_1\), and later in \(\sigma_2\). The dilemma we face here is whether to move the item in \(\sigma_1\) to the beginning, or not to move it at all.

**Lemma 3.3.2.** \textit{OPT\_FREE} moves \(x_i\) to the front of the list during \(\sigma_1\).

**Proof.** We prove this by induction over \(i\) starting from \(i = k + 1\).

In \(\sigma_2\), \(x_i\) has 3 consecutive requests. Thus, from lemma 1, \textit{OPT\_FREE} moves it to the head of the list during \(\sigma_2\). Thus, if we move \(x_i\) to the front early, in \(\sigma_1\), it saves us costs for serving \(\sigma_2\), when accessing this item. It saves us one unit cost for each predeceased item, which sums up to \(k\) units cost. For \(i = k + 1\) it does not matter for other items, since they are after \(i\) anyway, and so the order between them does not change. Thus, \textit{OPT\_FREE} only gains from moving \(x_{k+1}\) to the front during \(\sigma_1\), hence it will do so to retain its optimality.

Now, we assume the \textit{OPT\_FREE} moves \(x_j\) to the front for every \(k + 1 \leq j < i\), and we prove this for \(x_i\). We still gain \(k\) units for the first \(k\) items, as described in the base case. However, moving \(x_i\) to the front during \(\sigma_1\) also adds costs when accessing items preceding \(x_i\) in \(\sigma_1\), which are items \(x_j\) having \(k + 1 \leq j < i\). So, for the first \(k\) items in the list it saves us one unit cost per item, and so a total cost of \(k\). Let us look at other items preceding \(x_i\), that is, items \(x_j\) such that \(k + 1 \leq j < i\). The projection of the sequence \(\sigma_1\sigma_2\) over the items \(x_i, x_j\) is \(\sigma_{12,ij} = x_i x_j x_i x_j x_i\). From the inductive assumption, every such item is moved to the front of the list while servicing \(\sigma_1\), prior to any additional requests for \(x_i\). So, we pay only one additional unit cost when serving the first request to access \(x_j\) in \(\sigma_1\), and then the order of the two items is back to original. So, we pay one additional unit cost for each such item, which sums up to a total cost of \(i - k - 1\). In total, we found that by moving \(x_i\) to the front we save a cost of \(k\), but incur an additional cost of \(i - k - 1\). Since \(i \leq 2k\) clearly the optimum would be to move \(x_i\) to the front. 

3.3.3 Calculating the Cost of OPT_FREE

First we access the items between \(k + 1\) and \(2k\). We move them to the front. So when accessing each of them, it is located at position \(2k\) in the list. So the cost of accessing each of them is \(2k\) and for all of them together we obtain \(2k^2\). As for \(\sigma_2\), when accessing the first \(k\) items 3 times each, we incur a cost of \(\sum_{i=1}^{k} i + 2k = \frac{k(k+1)}{2} + 2k\). The first term accounts for the first access to each item, which is then moved to the front of the list.
The second term accounts for the two remaining accesses, with a cost of one each, for a total of \( k \) items. In total, we obtain \( OPT - FREE(\sigma) = 2k^2 + \frac{k(k+1)}{2} + 2k = 2.5k^2 + 0.5k \).

### 3.4 Calculating the Maximal Ratio

We now take \( k \) to infinity, so we neglect terms of lower order in \( k \), and leave only terms with square order in \( k \). By doing so, we obtain \( OPT(\sigma) = 2k^2 \) and \( OPT - FREE(\sigma) = 2.5k^2 \).

From here we obtain the ratio: \( \frac{FREE(\sigma)}{OPT(\sigma)} = \frac{2.5k^2}{2k^2} = 1.25 \)
Bibliography


[RW90] Nick Reingold and Jeffery Westbrook. Randomized algorithms for the list update problem. Technical report, Yale University, Department of Computer Science, 06 1990.


Boris Teia. A lower bound for randomized list update algorithms. 

האלגוריתם שאינה רשה לחלופים בחליפים בטליפה, אך אויה פושטת יוצר. לה, הוכחה ומכינה את הקקך עבורי ההוכחה הקשה יוצר, והדרשה עבורי האפסימסי של האלגוריתם שמסרתי רכ.
בחליפים חומימם.

结

תואנה הוא ש‚בשת מס飲 תועים רואית, כיSSH את אפסים על יסליו טובי מר’5/4 עלعال’,
נידרשק בבחוך לששנתלאלוגרמיים המצות בחליפים בטליפה. שיגיר, emerging מהשלום על העינה
סכלחליפים בטליפה. תואנה הוא ממלדה והנה על הדכחים בו של modeleים האפשריםיילעדים
 PTSטרופ מוסרת. ולא שלישים בחליפים בטליפה כحلولו מונגבל, ואתה הקרות שישים עליש

סהמה עלידי. 5/4
We can solve the original problem. The primal solution is a feasible solution of the primal problem, and the dual solution is a feasible solution of the dual problem. The value of the primal problem is always greater than or equal to the value of the dual problem, and the value of the dual problem is always less than or equal to the value of the primal problem. Therefore, the difference between the optimal value of the primal problem and the optimal value of the dual problem is a lower bound for the optimal value.

In the optimal primal, 1 is optimal. 1.6 is optimal COMB. If 1 is optimal, 1.6 is optimal COMB. If the gap is large enough, we cannot prove that 1.6 is optimal in the dual problem.

Our work presents a new result that addresses lower bounds for the list update problem, and the relation between blind moves and local moves. In every algorithm that uses local moves but does not use blind moves, it achieves the same value as the original algorithm. However, the converse is not true, as in the worst case, local moves are necessary for all algorithms.

In this work, we prove a lower bound for the optimal value of the algorithm presented. In addition, we analyze the cost of the algorithms for a single blind move and a local move.

To prove the lower bound, we need to prove the optimality of the presented algorithm. To prove the optimality of the presented algorithm, we need to show that no algorithm that uses only local moves can achieve a lower bound.

Technion - Computer Science Department - M.Sc. Thesis - MSC-2016-08 - 2016

iii
In the case of a large query, the single unit does not change. If the query is made for the first time, the algorithm moves the first unit to the front. If there are many changes, the algorithm can move the first unit to any position in the list, without limit, but the changes are made to the last unit. When there are changes in the list, the algorithm can move the first unit to any position in the list at any time.

There are classical algorithms for the list update problem, and one of them is the algorithm $x$.$\text{Move-To-Front}(x)$, or the algorithm $\text{MTF}$, which exchanges the list from the head to the tail. The algorithm $\text{2-Swap}$ is a non-deterministic algorithm, as it changes the first unit to any position in the list at any time. The algorithm $\text{1-Shuffle}$ is non-deterministic, as it exchanges the first unit to any position in the list at any time.

In addition, a non-deterministic version of the problem can be thought of as a version of the list where the head is always fixed. In this case, the algorithm can move the first unit to any position in the list at any time, while using the first unit is always the same. The non-deterministic version is more difficult.

The results of this work are significant. The first result is a new proof for the non-deterministic $\text{MTF, BIT, TS, COMB}$ algorithm for the classical algorithms. The algorithm $\text{DUAL FITTING}$ was used to find a linear algorithm for the problem. The algorithm $\text{DUAL FITTING}$ was used to find a linear algorithm for the problem. The algorithm $\text{DUAL FITTING}$ was used to find a linear algorithm for the problem. The algorithm $\text{DUAL FITTING}$ was used to find a linear algorithm for the problem. The algorithm $\text{DUAL FITTING}$ was used to find a linear algorithm for the problem.
תקציר

านהיה שונים ענפים, אשר שיכת עלולות באלגוריתמים שלמים, ונ 일을 תחומי
MTF, BIT, TS, COMB

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם

והם
המחקר בוצע בתנאי של פרופסור ספי נאור, בכוללת חדש המשנה.
בעיית עדכון הרשימה

היבנה על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מניסיון למעידים מבдуים המוחשב

ארז תמונת

фессל לוטננט טכניו – מכון טכנולוגי לישראל
ניח שטשRegExp חיפת 2016
בעיית עדכון הרשימה

אזר תמונת