ON THE LIST UPDATE PROBLEM WITH UNIFORM COSTS

PARSA MERAT

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

SEPTEMBER 2026

©PARSA MERAT, 2026

Abstract

The list update problem is a fundamental problem in online algorithms and data structures, concerning the maintenance of a linear list under a sequence of online access requests. In the standard setting, after accessing an item, it may be moved toward the front using a ***free exchange***, while adjacent items may be swapped using a ***paid exchange*** at unit cost. The goal is to minimize the total costs.

This thesis focuses on the ***uniform cost model***, where free exchanges are not allowed. Previous work established a lowerbound of 1.5 [1] on the ***competitive ratio*** while the best known upperbound is $\sqrt{7} \approx 2.646$ [2]. We establish a new 2.5 lowerbound for randomized algorithms under the partial cost model and present a $34/13 \approx 2.616$-competitive algorithm under both full and partial costs. We also give an optimal 2.5-competitive algorithm for lists of size 2, 3, and 4.

Table of Contents

Abstract ii

Table of Contents iii

List of Tables v

List of Figures vi

1 Introduction 1

1.1 Overview 1

1.2 Contributions 4

2 Literature Review 5

2.1 The Standard Model 5

2.2 The Uniform Cost Model 16

2.3 Other Cost Models 19

2.4 Relaxing the Online Constraint 25

2.5 Metrical Task Systems (MTSs) 27

3 Preliminaries and Problem Statement 29

3.1 Problem Statement 29

3.2 Analysis Tools 31
3.2.1 Comeptitvie Ratio 31
3.2.2 Potential Function Method 32
3.2.3 Projective Property 34

4 Randomized Lower Bound **38**
4.1 Overview 38
4.2 A Structural Lemma 39
4.3 The Main Result 42

5 A Randomized Algorithm **46**
5.1 Overview 46
5.2 Algorithm AltComb 49
5.3 Analysis of AltComb 50

6 Optimal Randomized Algorithm for Short Lists **58**
6.1 Overview 58
6.2 Algorithm BiPolar 59
6.3 Analysis of BiPolar 62
6.4 A Remark for Lists of Size $\ell \geq 5$ 76

7 Concluding Remarks **77**

Bibliography **80**

List of Tables

Table 4.1 Sequence probabilities and costs for Lemma 4.2.1 (assuming initial ordering of $x \to y$). 41

Table 4.2 Block probabilities and costs for Theorem 4.3.2 (assuming initial ordering of $x \to y$). 44

Table 5.1 Costs of algorithms over each phase. 51

Table 5.2 BIT's list configurations and costs over the sequence $(yx)^i$ with initial list ordering $x \to y$. After the first yx request, the configurations repeat every two requests. 54

List of Figures

Figure 6.1 An illustration of the case analysis of BiPolar for lists of length 3. Each of the five subfigures indicates one possible IO graph (directed graphs) along with the LS graph and consistent states before a request (solid enclosed areas) and after a request (dotted enclosed areas). 74

Figure 6.2 The List State (LS) graph for lists of size 4. 76

Chapter 1

Introduction

1.1 Overview

Efficient data organization and retrieval are fundamental concerns in computer science. Many applications, ranging from database systems and web search engines to operating systems and compilers, rely on maintaining collections of items that can be accessed quickly in response to user requests. As these systems process large numbers of queries over time, even small improvements in access efficiency can lead to substantial gains in overall performance. This motivates the study of self-organizing data structures, which dynamically adapt to observed access patterns to reduce future access costs.

A classical abstraction of this setting is the ***dictionary problem***. A dictionary maintains a set of items and supports three basic operations:

- **Insert**: add a new item to the dictionary,
- **Delete**: remove an existing item from the dictionary,
- **Search**: locate a specified item.

The goal is to implement these operations as efficiently as possible. Numerous data structures have been proposed for this purpose, including balanced search trees, hash tables, and self-adjusting structures. In many applications, search operations occur much more frequently than insertions and deletions. Consequently, understanding how to optimize repeated searches becomes a central algorithmic challenge.

One particularly simple model for the dictionary problem is a ***dynamic list*** which has been studied for more than six decades [3, 4]. In this model, the dictionary is represented as an unsorted linear list containing all items. To search for an item, the list is scanned sequentially from the front until the desired element is found. The cost of the search is therefore proportional to the position of the item in the list. After an access, the list may be reorganized according to some update rule in an attempt to improve the efficiency of future searches. Examples include moving the accessed item to the front of the list or exchanging it with neighboring elements.

The resulting optimization problem is known as the ***online list update problem***. A sequence of requests arrives one at a time, and after serving each request, the algorithm may perform list rearrangements without knowledge of future requests. The objective is to minimize the total access and reorganization cost incurred over the entire request sequence. Because decisions must be made before future requests are known, the list update problem belongs to the broader class of ***online problems***.

In an ***online*** problem, the input is revealed incrementally, and an algorithm must make decisions based only on information available up to the current time. This contrasts with the traditional ***offline*** setting, where the entire input sequence is known in advance. Since an online algorithm cannot access future requests, it is generally impossible for it to achieve the same performance as an algorithm with

complete knowledge of the future. The central challenge is therefore to design online algorithms that perform well despite this informational disadvantage.

The standard framework for evaluating online algorithms is ***competitive analysis*** [5]. Since an online algorithm must make decisions without knowing future requests, its performance is typically compared to that of an idealized offline algorithm that has complete knowledge of the entire request sequence in advance. Competitive analysis measures how much worse the online algorithm can perform relative to this omniscient benchmark in the worst case. Informally, a deterministic online algorithm is said to be ***c-competitive*** if, on any request sequence, its total cost is at most a constant factor c times the cost of the optimal offline algorithm (often denoted as OPT), up to lower-order additive terms. The competitive ratio for a randomized algorithm is defined similarly, except that its cost is measured in terms of ***expected total cost***. Thus, a smaller competitive ratio indicates that the online algorithm performs closer to the best possible strategy despite having no knowledge of future requests. This framework has become one of the primary tools for studying online algorithms and has played a central role in the development of the theory of list update algorithms.

Although the dictionary problem includes insertions and deletions, much of the theory of list update focuses exclusively on search operations. This restriction is justified for several reasons. First, in many practical settings, searches vastly outnumber updates, making search cost the dominant contributor to overall performance. Second, insertions and deletions can often be incorporated into existing analyses with relatively minor modifications, whereas the primary algorithmic difficulty arises from adapting the list to changing search patterns. Finally, concentrating on searches isolates the essential trade-off between access cost and reorganization cost, allowing

researchers to study the behavior of online algorithms in a cleaner setting.

For these reasons, the standard formulation of the online list update problem assumes a fixed set of items and considers only a sequence of search requests. The challenge is then to design online reorganization strategies whose performance is as close as possible to that of an optimal offline algorithm. This problem has become a central topic in the study of online algorithms, leading to a rich body of work on competitive analysis. In the standard list update problem, upon accessing an item x in the list, it can be moved anywhere closer to the front of the list at no cost; this is referred to as a ***free exchange***. In reality, however, this may not be practical, and an alternative model in which free exchanges are not allowed is presented and studied in several works [1, 2, 6]. Our focus in this thesis is on the latter model, often referred to as the ***uniform cost model***.

1.2 Contributions

In this thesis, we focus primarily on randomized algorithms in the uniform cost model, as explained in detail in Chapter 3. In Chapter 4, we improve the best-known lower bound on the competitive ratio from 1.5 of [1] to 2.5 (Theorem 4.3.2). In Chapter 5, we present a new upper bound of $34/13 \approx 2.616$ via a new randomized online algorithm (Theorem 5.3.1), improving on the previous best upper bound of 2.646 [2]. Finally, in Chapter 6, we characterize an optimal randomized algorithm for lists of sizes 2, 3, and 4 and provide a particularly elegant proof of its optimality via a potential-function argument, which is expected to be useful in the analysis of other list-update algorithms.

Chapter 2

Literature Review

The list update problem is a classical problem in online algorithms. It was first studied by McCabe [3] in the context of serial files with relocatable records, with the analysis focusing on expected access costs under probabilistic models and independent request probabilities. Much of the subsequent work continued to study the problem from a probabilistic perspective. A major shift came with the seminal work of Sleator and Tarjan [5], which introduced competitive analysis to the list update problem and established the standard online model. In this model, requests arrive sequentially, and an online algorithm must serve each request without knowledge of future requests.

2.1 The Standard Model

In the standard list update problem [5], an algorithm maintains an ordered list of ℓ items that is accessed via n sequential requests. Each request must be served before the next one arrives. To process a request, the algorithm linearly scans the list from the head until the requested item is found, and the cost of this operation is equal to

the position of the item in the list at that time. After serving a request and before the next one arrives, the algorithm is allowed to reorganize the list. It may move the requested item to any position closer to the front using a ***free exchange***, which incurs no cost. Additionally, it may swap any two adjacent items using a ***paid exchange***, each at unit cost. The idea behind free exchanges is that moving an accessed item forward requires only a constant number of pointer updates, making it inexpensive compared to paid reordering operations.

Partial vs Full Cost Models. Two common cost models used in competitive analysis are the full cost model and the partial cost model, which differ in how they charge for accessing list elements. In the full cost model, accessing an element at position i incurs a cost of i. In contrast, the partial cost model assigns the access cost as $i - 1$, effectively not charging for the step of accessing the element itself. Although the difference between the two models is only a constant shift per access, it can significantly simplify theoretical analysis.

The partial cost model is particularly useful because it aligns well with ***projective*** algorithms. Projective algorithms are a class of algorithms whose behavior on a list of arbitrary size can be fully characterized by reducing the analysis to all induced subproblems on lists of size 2. In other words, rather than analyzing the global evolution of the entire list, one can decompose the algorithm's performance into pairwise interactions between elements. Each such interaction captures how the algorithm preserves or changes the relative ordering of two items over time, and the overall cost can be expressed as the aggregation of these local contributions across all pairs. Consequently, to establish an upper bound for the competitive ratio, it suffices to

analyze the algorithm's behavior restricted to lists of size 2, since these pairwise cases completely determine the performance on larger lists.

In what follows, all results apply to both the partial and full cost models unless explicitly stated otherwise.

Deterministic Algorithms

Move-to-Front (MTF). One of the earliest and most influential algorithms is Move-To-Front (MTF). Under MTF, whenever an item is requested, it is moved immediately to the front of the list using a free exchange. In a seminar paper on the theory of online algorithms, Sleator and Tarjan [5] showed that MTF is 2-competitive, i.e., its cost is twice that of an optimal offline algorithm in the worst case. They also proved that no deterministic online algorithm can achieve a competitive ratio better than 2 for sufficiently long lists. To see this, consider all possible ***static*** list orderings. Averaging the cost over all permutations, each item appears in each position equally often. Thus, for any request sequence, there exists a static ordering with average access cost at most $\ell/2$ per request. This gives an upper bound of $n\ell/2$ on the optimal offline cost for a sequence of length n. In contrast, in the online setting, an adversary can force any deterministic algorithm to incur a worst-case cost of at least ℓ per request by repeatedly requesting the last item in the algorithm's list, for a total cost of $n\ell$. Consequently, no deterministic online algorithm can achieve a competitive ratio better than $n\ell/(n\ell/2) = 2$. This result establishes the optimality of MTF among deterministic algorithms.

TimeStamp (TS). Besides MTF, several other online self-organizing list algorithms have been proposed that achieve an optimal competitive ratio of 2 under the standard model. One such algorithm is TIMESTAMP(TS) [7]: After an item x is accessed, TS algorithm moves x forward in the list until it is positioned immediately before the first item y in the list that precedes x and has been requested at most once since the previous access to x. If no such item y exists, or if x has never been accessed before, TS leaves the list unchanged and performs no reordering.

As pointed out, this deterministic algorithm has an (optimal) competitive ratio of 2.

Example 2.1.1. *Consider the list $A \to B \to C \to D$. And the request sequence of: D, C, B, A, A, C*

For the first 4 requests of D, C, B, A, since this is the first access to each item, no reordering occurs and the list remains unchanged.

The next request is A. According to TS, A is moved in front of the first preceding item that has been requested at most once since the previous access to A. Since there have been no requests between the two accesses to A, it is moved to the front of the list. However, A is already at the front, so the list remains unchanged.

The next request is C. The items preceding C in the list are A and B. Since the previous access to C, item A has been requested twice, while item B has been requested once. Therefore, the first item in the list that precedes C and was requested at most once is B, therefore C is moved in front of B. The resulting list is

$$A \to C \to B \to D.$$

Other Optimal Online Algorithms. Following the same intuition as TS, El-Yaniv showed that a rich class of online list update algorithms achieves optimal competitive ratios [8]. In particular, he introduced a family of algorithms called Move-to-Recent-Item (MRI). In this family, each algorithm is parameterized by an integer $k \geq 1$. When an item x is accessed, MRI inserts x immediately after the last item y preceding x in the list that has been requested at least $k+1$ times since the last access to x. If no such item y exists, or if x is being accessed for the first time, the algorithm simply moves x to the front of the list. This can be viewed as a generalization of TS. In TS, the requested item is placed immediately after the last item that is requested at least twice since its previous access (an equivalent formulation of TS). In MRI, the threshold of two requests is replaced by a parameter, allowing the reordering rule to be adjusted according to the chosen integer value. Note that when parameter $k = 1$, MRI is equivalent to TS. At the other extreme, when $k = \infty$ every requested item is moved to the front of the list, and thus MRI becomes equivalent to MTF.

In a similar line of work, Schulz proposed another family of 2-competitive algorithms, called Sort-by-Rank (SBR), which is also parameterized by $0 \leq \alpha \leq 1$ and reduces to MTF and TS at the extreme values of α.

Other Popular Algorithms. Several other deterministic strategies have also been investigated. The TRANSPOSE algorithm swaps the accessed item with the item immediately preceding it, gradually moving frequently requested elements toward the front (and thus it is practical in certain applications). This algorithm is clearly not competitive; in particular, its competitive ratio grows with the list length ℓ.

For sequences that repeatedly request the last item in the list, the algorithm keeps swapping the last two items and never moves them closer to the front. The cost of TRANSPOSE would be $n\ell$ while an optimal algorithm can move those two items to the front of the list and pay an amortized cost of 1.5 per access, yielding a total cost of $1.5n$ and thus the competitive ratio for TRANSPOSE will be at least $\frac{n\ell}{1.5n} = 2\ell/3$

Another classical heuristic is FREQUENCY COUNT, which maintains items in non-increasing order of access frequency. This algorithm is oblivious to the order of items in the input sequence and performs poorly on inputs with long runs of repeated items. For example, consider a sequence starting with $n/2 + 1$ 'a's followed by $n/2 - 1$ 'b's (for a large even integer n). In this input, Frequency Count never places b before a in the second half of the input. Extending this input from two items to ℓ items yields a worst-case sequence in which the competitive ratio of Frequency Count grows with ℓ.

To summarize, although algorithms like TRANSPOSE and FREQUENCY COUNT often perform well empirically under certain request distributions, they do not achieve constant competitive ratios in the worst case. This highlighted the distinction between practical average-case behavior and worst-case guarantees under competitive analysis.

In this thesis, we are mainly concerned with worst-case guarantees provided by competitive analysis.

Randomized Algorithms

As discussed previously, every deterministic list update algorithm has a competitive ratio of at least 2. To surpass this lower bound, several randomized algorithms have been proposed. In the randomized setting, adversaries are classified according to the amount of information they possess about the algorithm's random choices.

- An "oblivious" adversary has no knowledge of the algorithm's random bits. Equivalently, it must fix the entire request sequence in advance, before the algorithm begins processing the sequence.

- An "adaptive online" adversary is more powerful: it generates requests incrementally, choosing each request after observing the algorithm's responses to all previous requests.

- An "adaptive offline" adversary is stronger still, as it has complete knowledge of the algorithm's random bits. Consequently, it can predict in advance how the algorithm will behave on any request sequence before presenting that sequence to the algorithm.

Reingold et al. [2] showed that adaptive online and adaptive offline adversaries are equally powerful. In fact, they proved a lower bound of 2 on the competitive ratio of any randomized algorithm against adaptive adversaries. Since this matches the best achievable ratio for deterministic algorithms, randomization does not improve the performance against adaptive adversaries. Consequently, an optimal deterministic algorithm (e.g., MTF) is also optimal against both adaptive online and adaptive offline adversaries. From this point onward, we therefore consider randomized algorithms only in the setting of an oblivious adversary.

Randomized Move-To-Front (RMTF). The Randomized Move-To-Front (RMTF) algorithm is a randomized variant of MTF. Upon serving a request, RMTF moves the requested item to the front of the list with probability 1/2; otherwise, the list remains unchanged. RMTF is known to be at least 2-competitive [9]. Consider

a request sequence that repeatedly requests the item initially at the end of the list, then repeatedly requests the item initially in the second-to-last position, and so on. Suppose each item is requested sufficiently many times before proceeding to the next one. For an item initially at position ℓ, RMTF incurs an expected cost of

$$\ell(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots) = 2\ell,$$

since after each request the item is moved to the front with probability 1/2. In contrast, OPT can move the item to the front after its first access and thereafter serve all remaining requests to that item at unit cost, yielding a total cost of approximately ℓ for sufficiently long phases. Thus, the ratio between the expected cost of RMTF and the cost of OPT approaches 2, implying that RMTF is at least 2-competitive.

The power of randomization in RMTF is essentially wasted, since the deterministic MTF algorithm already achieves a 2-competitive ratio. RMTF does not improve this, but only adds unnecessary randomness to the same strategy.

SPLIT. Irani [10] introduced the first randomized algorithm that surpasses the deterministic lower bound. The algorithm, known as SPLIT, assigns to each element x a pointer to another element that appears earlier in the list. Upon accessing x, the algorithm moves it either to the front of the list or just ahead of the element indicated by a pointer assigned to x, depending on the outcome of a random choice; the exact pointer update rules are not needed for our discussion here. This randomized approach attains a competitive ratio of 1.875 [10].

Bit. Reingold et al. [2] introduced another randomized algorithm called BIT. Prior to processing the request sequence, each item x is assigned a bit $b(x)$, independently chosen to be either 0 or 1 with equal probability. When x is accessed, its bit is flipped. After this update, the algorithm moves x to the front of the list if the new value of $b(x)$ is 0; otherwise, if $b(x) = 1$, the list order remains unchanged. Intuitively, this can be viewed as moving each element to the front on every other access, while the initial random bits serve only to randomize the starting configuration of the bits of the algorithm. Reingold et al. [2] use the potential function method to show that BIT is 1.75-competitive.

Counter Family. BIT belongs to a broader class of online algorithms known as COUNTER algorithms [2]. A COUNTER algorithm is defined by two parameters: an integer s and a fixed subset $S \subseteq \{0, 1, \cdots, s-1\}$. For each item, the algorithm maintains a counter modulo s. Upon accessing an item x, its counter is decremented (or reset to $s-1$ if it is 0). If the updated counter value lies in S, then x is moved to the front of the list. By choosing appropriate values of S and s, the resulting COUNTER algorithm achieves a competitive ratio that converges to $\sqrt{3} \approx 1.732$ [2].

Random Reset Strategy. RANDOM RESET [2], similar to COUNTER, maintains for each item a counter modulo s, which is decremented (or reset if it is 0) upon every access. It moves an accessed item to the front of the list whenever the updated counter is 0; otherwise, it performs no reordering. After a move-to-front operation, the counter for that item is reset to a value j chosen randomly with respect to a probability distribution π_j. By choosing an appropriate resetting distribution and value of s, RANDOM RESET achieves a competitive ratio of $\sqrt{3} \approx 1.732$. RANDOM

RESET aims to improve upon COUNTER by incorporating additional randomness. While COUNTER uses an arbitrarily large parameter s (and consequently large counter values for each item) to approach the $\sqrt{3}$ competitive ratio, RANDOM RESET instead employs smaller counter values and increased randomness to achieve the same result.

Barely Random Property. A notable property of both BIT and COUNTER is that they are *barely randomized* algorithms: they use only a constant number of random bits during initialization, after which their execution is fully deterministic. In contrast, RANDOM RESET requires $\Omega(n)$ random bits, where n denotes the length of the input sequence.

Randomized Timestamp. The deterministic TS algorithm discussed earlier can be viewed as a special case of the RANDOMIZED TS(p) algorithm [7]. In this scheme, each request is processed by applying MTF with probability p, and the deterministic TS rule with probability $1 - p$. Clearly, the deterministic TS is the special case with $p = 0$. Choosing an appropriate value of p yields a competitive ratio of approximately 1.618 (the golden ratio).

Comb. The COMB algorithm [11] combines the deterministic TS strategy with BIT. At the beginning, COMB randomly selects one of the two strategies: with probability 0.8, it selects BIT, and with probability 0.2, it selects the deterministic TS algorithm. Consequently, it proceeds with applying the selected algorithm to serve all requests. This hybrid approach achieves a competitive ratio of 1.6, which is the best known randomized upper bound against an oblivious adversary to date.

Lower Bounds. There have also been several works on establishing lower bounds for the competitive ratio of randomized list update algorithms [2, 10]. The strongest known general lower bound is 1.5, due to Teia [12]. Furthermore, Ambühl et al. [13] proved a lower bound of 1.50084 for randomized online algorithms in the partial cost model, where accessing an item in the i-th position incurs a cost of $i - 1$.

For the class of projective algorithms, which includes most of the algorithms discussed so far, a stronger lower bound of 1.6 is known [14], implying that COMB is optimal within this restricted class. Despite these results, a gap remains between the best-known upper bound of 1.6 and the lower bound of 1.5 (or 1.50084 in the partial-cost model) for the list update problem.

Small Lists. Unlike the general setting, the status of the problem is well understood for smaller lists. In particular, for lists of size at most 4, there exist Markov chain algorithms that achieve the competitive ratio of 1.5, and this is the best possible ratio attainable in this setting [15].

Are Paid Exchanges Necessary? Most of the online algorithms proposed for the standard model rely exclusively on free exchanges. The intuition is that when an item is requested, it should be moved closer to the front of the list because it is likely to be requested again. Sleator and Tarjan suggested that free exchanges alone might be sufficient [5] for an optimal offline algorithm. However, Reingold and Westbrook showed that there exist request sequences for which any optimal algorithm must move an item toward the back of the list, implying that paid exchanges are sometimes necessary for optimality [16]. Furthermore, López-Ortiz et al. [17] show a lower bound of 12/11 on the ratio between the cost of an optimal offline algorithm

restricted to free exchanges and that of an optimal offline algorithm that is allowed to perform both paid and free exchanges. This implies that paid exchanges are necessary, even in the offline setting, in order to achieve optimality.

Reingold and Westbrook [16] demonstrated that optimal offline algorithms can be transformed to use only paid exchanges. Specifically, if an algorithm performs a free exchange after serving a request, an equivalent offline algorithm can instead execute the corresponding paid exchanges before the request (more precisely, during the handling of the previous request, since an algorithm must first access the requested item before performing any exchanges). The overall cost remains unchanged because the cost of these paid exchanges is offset by the reduced access cost of the subsequent request.

2.2 The Uniform Cost Model

The standard model has been criticized in the literature for failing to accurately capture the true costs incurred in certain implementations [2, 18, 19], particularly because of the assumption that certain swaps can be performed for free.

A natural way to address the problem of free exchanges is to disallow free exchanges and charge a cost of 1 for any exchange of adjacent items as before. Since free exchanges are not permitted, algorithms that rely heavily on them are significantly affected. In particular, MTF and TS both attain a (suboptimal) competitive ratio of 4 under this model.

Deterministic Algorithms. The best current deterministic algorithm is Full-Or-Partial-Move (FPM) [6]. FPM maintains a target index for each item in

the list. Upon each request, the algorithm either moves the requested item to the front (a full move) or only to its designated target position (a partial move). The choice between these two actions is guided by a potential-function argument that balances access costs against reorganization costs, yielding a competitive ratio of at most 3.3904. In essence, the algorithm is heavily attached to its analysis.

On the lower bound side, no deterministic algorithm can achieve a competitive ratio better than 3 [2]. The proof is given via an elegant argument, which we include here for completeness.

Theorem 2.2.1. *[2] Under the uniform cost model, no deterministic algorithm for the list update problem can achieve a ratio better than 3.*

Proof. Construct the input sequence σ as follows. At each time step, the adversary requests the last element (i.e., the element at position ℓ) in the list maintained by the algorithm ALG. Thus, each access incurs a cost of ℓ, and the total cost incurred by ALG on the request sequence σ, denoted by $\text{ALG}(\sigma)$, is $n\ell + D$, where D is the total cost of paid exchanges performed by ALG.

We compare $\text{ALG}(\sigma)$ against two offline benchmarks.

- First, consider OFF-FREQ, the offline optimal algorithm that orders the list by non-increasing frequency of the full request sequence. The worst-case cost of OFF-FREQ happens when all the items have the same frequency. Therefore, we have $\text{OFF-FREQ}(\sigma) \leq \frac{n\ell}{2} + \alpha$, where α is a constant that accounts for the cost of paid exchanges at the start to reorder the list.

- Second, define $\overline{\text{ALG}}$ as the offline algorithm that maintains the reverse ordering of ALG's list at every time step. Each access by ALG accesses an element at the

end of its list, while in $\overline{\text{ALG}}$ the same element is at the front, so every access for $\overline{\text{ALG}}$ costs 1. On the other hand, after each access, $\overline{\text{ALG}}$ pays the same cost as ALG to apply paid exchanges in order to keep its list in the reverse order of ALG's list. There is also an extra $\ell(\ell-1)/2$ cost at the start to reverse the initial list. We conclude that $\overline{\text{ALG}}(\sigma) = n + D + \ell(\ell-1)/2$.

We now proceed to bound the competitive ratio of ALG, denoted by $R(\text{ALG})$. We consider two cases, depending on the value of D (the number of paid exchanges for ALG), and exploit one of the upper bounds for OPT for each of the two cases.

- If $D \geq \frac{n\ell}{2}$ we use $C_{\text{OFF-FREQ}}$ as an upper bound on the cost of OPT:

$$R(\text{ALG}) \geq \frac{\text{ALG}(\sigma)}{\text{OFF-FREQ}(\sigma)} = \frac{n\ell + D}{n\ell/2 + \alpha} \approx \frac{n\ell + D}{n\ell/2} = 2 + \frac{2D}{n\ell} \geq 3$$

- Otherwise, when $D \leq \frac{n\ell}{2}$, we use $\overline{\text{ALG}}(\sigma)$ instead as an upperbound on the cost of OPT.

$$R(\text{ALG}) \geq \frac{\text{ALG}(\sigma)}{\overline{\text{ALG}}(\sigma)} = \frac{n\ell + D}{n + D + \ell(\ell-1)/2} \approx \frac{n\ell + D}{n + D} \geq \frac{3\ell}{\ell + 2} \approx 3$$

Thus, in both cases,

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} \geq 3,$$

showing that the competitive ratio of ALG is at least 3.

□

Randomized Algorithms. For randomized algorithms, a member of the COUNTER family [2] achieves a competitive ratio of 2.75. By using significantly more space, the competitive ratio can be made arbitrarily close to $\sqrt{7} \approx 2.646$. Although additional space is typically not a primary concern in competitive analysis, they also showed that a member of the RANDOM RESET family [2] achieves $\sqrt{7} \approx 2.646$-competitiveness (at the cost of using $\Omega(n)$ random bits instead of being a barely random algorithm).

On the lower bound side, it is known that no randomized algorithm can achieve a competitive ratio below 1.5 [1].

2.3 Other Cost Models

In this section, we discuss several other models that have been studied in the literature.

The Paid Exchange (P^d) Model

The paid exchange (P^d) model [2, 20] is a generalization of the uniform cost model. Under this model, free exchanges are not allowed, and each paid exchange incurs a cost of $d \geq 1$. The special case of $d = 1$ is the uniform cost model. This model is motivated by the observation that swap operations typically require significantly more time than a single search step, because write operations are generally more expensive than reads. As a result, moving an accessed item closer to the front of the list is treated as a costly operation.

Since the competitive ratios of most algorithms for various values of d are bounded

between two extremes defined by $d = 1$ (uniform cost model) and $d \to \infty$, we are particularly interested in the regimes of $d = 1$ and $d \to \infty$. In the latter case of $d \to \infty$, the best-known deterministic competitive ratio is achieved by a determenistic variant of the COUNTER algorithm, in which the item counters are all initialized to zeros. This algorithm is 4.56-competitive [2]. It should be noted that, using the same proof as [2] (see Theorem 2.2.1), no deterministic algorithm can achieve a competitive ratio better than 3 for ***any*** value of d.

For randomized algorithms in the paid exchange model with $d \to \infty$, a member of COUNTER family achieves a competitive ratio of 2.2808 [2], while the current best known upper bound is given by the TS(ℓ, p) algorithm, which is 2.2442-competitive [1]. On the lower-bound side, a bound of 2 is known in the partial cost model [1], and the strongest known lower bound in the full cost model is 1.8654 [1].

Exchanges in Between One important point is that in [1], the authors consider a variant of the paid exchange model in which the underlying cost structure differs from that of the standard paid exchange/uniform cost model. In particular, they allow exchanges ***both before and after*** serving a request, whereas in the other models, exchanges can occur only after serving. In particular, in the standard model, an algorithm does not know the location of the accessed item before incurring the access cost and thus cannot decide whether to pay to move it before accessing it using paid exchanges.

This "before serving" capability is the key distinction between the two models. For instance, when $d = 1$ (corresponding to the uniform cost model), in the model of [1], an item at position i can be moved to the front at cost $i - 1$ before access, and then

accessed at cost 1, effectively allowing free exchanges. As d increases, exchange costs become increasingly dominant relative to access costs, and their model's behavior approaches the standard P^d model.

The Linear Cost (MRM) Model

Although the online list update problem has been extensively studied within the traditional framework, the assumptions underlying its cost model have been called into question. Independently, Martínez and Roura [18] and Munro [21] argued that the standard cost model does not accurately represent the actual cost of list reorganization in practical settings.

Consider a list $a_1 \to a_2 \to \ldots \to a_\ell$ maintained by an online algorithm. In realistic implementations, rearranging the elements preceding an accessed item a_i (to form a different ordering) generally requires time proportional to i. However, within the standard model, the same operation (using a sequence of paid exchanges) may incur a cost proportional to i^2, leading to a significant overestimation.

To demonstrate this issue in more detail, Munro considered the operation of accessing the last element in a list and then reversing the entire list. In commonly used data structures such as arrays or linked lists, this operation can be performed in $O(\ell)$ time. By contrast, the standard model assigns an approximate cost of $\ell^2/2$ to the same operation. Based on this discrepancy, these studies argued that the traditional model limits online algorithms' ability to exploit efficient reorganization strategies. Consequently, they proposed an alternative cost model in which the combined cost of accessing the i-th item and reorganizing the first i elements remains linear in i.

This model has been studied as the MRM model [19, 22, 23]. An unexpected

consequence of adopting this more realistic cost model is that the offline optimum gains a significantly greater advantage from the free rearrangements allowed by the MRM model than an online algorithm does. In fact, under the MRM framework, it was shown that for certain arbitrarily long request sequences, every online algorithm incurs an amortized access cost of $\Theta(\ell)$ per request. In contrast, an optimal off-line strategy achieves a cost of only $\Theta(\log \ell)$ for any request sequence. As a result, all online list update algorithms exhibit a competitive ratio lower bounded by $\Omega(\ell/\log \ell)$. We also note that finding the optimal offline solution is no longer NP-hard in the MRM model (unlike the standard model), as it can be found in polynomial time $O(n^3)$ [19].

At first glance, these results may suggest that the modified cost model excessively favors the offline optimum and therefore gives it unrealistic computational power. However, the reorganization capabilities provided under this model reflect operations that can realistically be performed in practical systems. Therefore, the observed performance gap does not stem from an artificial enhancement of the offline algorithm but rather from a more accurate representation of the actual costs of list maintenance and rearrangement.

The Logarithmic Cost (Compression) Model

The list update problem admits a natural interpretation in terms of data compression [24]. In this framework, each symbol in the alphabet corresponds to an item in the list, and the text to be compressed is treated as a sequence of access requests. Given an input sequence (a text to be encoded), the compression mechanisms apply a deterministic algorithm ALG to encode the positions of items accessed by ALG in the compressed sequence while updating the list using ALG. For decoding, provided

with the sequence of access costs (compressed data), the original text can be uniquely reconstructed from the encoding of positions (indices) of accessed items while updating the list using the same algorithm ALG. Consequently, the total length of the compressed representation is proportional to the sum of the lengths of the encoded accessed positions.

In many realistic encodings, however, access costs are represented in binary. As a result, access to the i-th position in the list can be encoded using $\Theta(\log i)$ bits. This changes the effective cost of access from linear to logarithmic when analyzing compression efficiency. This observation leads to refined analyses of list update algorithms under more realistic encoding assumptions and has been used to derive tighter bounds and alternative interpretations of classical results [25, 26].

The Offline Problem

The offline problem for the standard model of list update was shown to be NP-hard [27], meaning that, in general, finding an optimal reordering strategy is computationally intractable. This result establishes the inherent difficulty of the problem under standard complexity assumptions.

An exact offline algorithm can be obtained in super-polynomial time by a straightforward adaptation of the offline algorithmic framework of Manasse, McGeoch, and Sleator for task systems [20], as noted by Reingold and Westbrook [16]. The method employs a dynamic programming formulation over all permutations of the list. Each state corresponds to a particular ordering of the ℓ elements, and transitions represent serving a request followed by an arbitrary reordering of the list. Since the state space consists of all $\ell!$ permutations, the dynamic program maintains a value for every con-

figuration and considers transitions between every pair of states after each request. This yields $\ell! \times \ell!$ possible transitions per request, and therefore a total running time of $\Theta((\ell!)^2 n)$ [16] over a request sequence of length n. Although exact, this approach is computationally infeasible in practice, as it operates over the full permutation state space without exploiting additional structure or redundancy in the problem.

Reingold and Westbrook [16] introduced the concept of ***subset transfers*** and showed that there exists an optimal offline algorithm that only uses this type of operation to adjust the list. A subset transfer, when accessing an element a_i, consists of selecting a subset of the elements preceding a_i and moving it directly to a position immediately behind a_i, while preserving the relative order of the moved elements. This restriction is significant because it shows that it is not necessary to consider arbitrary permutations of the list during reorganization; instead, optimal solutions can be represented using only these structured moves. As a result, the number of relevant transformations is reduced by considering all $\ell!$ possible reordering operations to a combination of subset choices together with restricted permutations on the remaining structure. This characterization leads to a dynamic programming formulation where states are indexed not only by permutations of the list but also by subsets of elements that have been "activated" through previous accesses. The resulting state space combines subset selection (giving a 2^ℓ factor) with ordering over the remaining elements (contributing a $(\ell - 1)!$ factor), yielding a total running time of $\Theta(2^\ell(\ell - 1)!n)$.

Divakaran [28] builds further on the subset-transfer characterization. They introduced an improved dynamic programming formulation that reduces the computational overhead of handling these subset-transfer states. By streamlining how permutations are updated and evaluated within each state, they reduce the per-transition

cost compared to earlier approaches, while keeping the same $(\ell - 1)!$ structural component. This yields an overall running time of $\Theta(\ell^2(\ell - 1)!n)$.

2.4 Relaxing the Online Constraint

The classical list update problem assumes a fully online setting, in which requests must be served immediately and with no knowledge of the future. A natural question is whether stronger performance guarantees can be obtained by relaxing this constraint and providing the algorithm with additional information or flexibility. Several models have been proposed in this direction. These models differ in the type and amount of information available to the algorithm, but all aim to bridge the gap between online and offline computation. We briefly review the main results for each setting below.

Lookahead. A natural way to relax the strict online nature of the list update problem is to provide the algorithm with limited information about future requests. One such relaxation is ***lookahead***, where the algorithm is allowed to observe the next k requests before making a decision. Unfortunately, for the list update problem, any reasonably small lookahead is useless in the worst-case scenarios. In particular, Albers [29] shows that in the standard deterministic model, a lookahead of size $\Omega(\ell^2)$ is necessary to achieve an additive constant improvement over the classical competitive ratio of 2.

Deadlines. Beyond standard lookahead, several alternative models have been proposed to incorporate partial future knowledge or auxiliary information. In particular, the ***windowed request*** and ***delayed service*** models do not need to search for the

requested item immediately; instead, each request is accompanied with a deadline by which it must be served (in the case of delayed service setting, a delay function that imposes a penalty is applied), thereby introducing flexibility in the timing of operations [30]. Azar et al. show that MTF is not competitive in these settings and present alternative algorithms achieving $O(1)$-competitiveness [30].

Advice. Another line of work considers the ***advice*** model, in which the online algorithm receives a limited number of bits of information from an oracle with full knowledge of the input sequence and unbounded computational power. Boyar et al. [31] studied the advice complexity of the list update problem, showing that while linear bits of advice is required for optimality, only two bits of advice suffice to obtain a deterministic $5/3 \approx 1.66$-competitive algorithm, improving upon the best possible competitive ratio of 2 without advice. The two advice bits are used to select the best-performing algorithm among Move-To-Front Odd (MTFO), Move-To-Front Even (MTFE), and TS for a given request sequence. Here, MTFO and MTFE are deterministic variants of Bit in which all bits are initially set to either 0 or 1, rather than chosen at random.

Prediction. More recently, the ***prediction*** model has been studied, in which the algorithm is given (possibly imperfect) predictions of future requests [32]. This setting captures modern learning-augmented approaches and allows for the analysis of performance guarantees under prediction error. Azar et al. [33] propose a $(1+\epsilon)$-smooth randomized algorithm with robustness $O(1/\epsilon^4)$. ***Smoothness*** ensures that the algorithm's cost is at most a factor $1+\epsilon$ larger than the cost of an algorithm that follows the predictions exactly. In contrast, ***robustness*** guarantees that, even

when the predictions are arbitrarily inaccurate, the algorithm's cost remains within a factor $O(1/\epsilon^4)$ of the optimal offline cost for any input sequence. The prediction used by the algorithm is the ordering of the list maintained by OPT after every request. They observe that when requests are drawn from a distribution, the list maintained by OPT is ordered by decreasing request frequency. After each request, this ordering is provided as the prediction for that step, since it may change over time.

2.5 Metrical Task Systems (MTSs)

A ***Metrical Task System*** (MTS) [34] is an online optimization framework consisting of a set of states and a metric cost function that determines the cost of transitioning between states. The system receives a sequence of requests, each of which assigns a processing cost to every state. Upon the arrival of a request, an online algorithm can choose to move to a new state and then serve the request. Therefore, the algorithm must decide how to process each request while minimizing both processing and transition costs. MTS is one of the most widely studied models in online algorithms and generalizes problems such as paging, list update, and the k-server problem [9].

Bounds. For the general metrical task systems, the deterministic competitive ratio is completely characterized: the optimal deterministic ratio is $2N - 1$ where N is the number of states in the system, and this bound is tight [34]. In the randomized setting, the best-known upper bound for arbitrary N-point metrics is $O(\log^2 N)$ [35]. Earlier work established lower bounds of $\Omega(\sqrt{\log N / \log \log N})$ [36]; however, more recent results strengthened this to an $\Omega(\log^2 N)$ lower bound [37], matching the known upper bound asymptotically.

List Update as an MTS Problem. As mentioned previously, the list update problem can be viewed as a special case of the MTS framework by interpreting each of the $N = \ell!$ permutations of the list as a state in a metric space. The distance between two states is defined as the minimum cost of transforming one permutation into another using adjacent swaps (i.e., the shortest-path distance in the corresponding permutation graph, which is equivalent to the Kendall tau distance [38]). Although MTS is typically described as moving first and then paying the service cost, this difference is only a matter of time indexing. In the list update problem, we pay the access cost before performing the reordering step, but in MTS, this is equivalent to evaluating the cost in the current state and then transitioning afterward. Thus, the two formulations are equivalent up to a shift in the indexing of the states.

Note that applying general MTS algorithms to this formulation would yield a deterministic competitive ratio of $2\ell! - 1$ and a randomized ratio of $\Theta(\log^2(\ell!))$, both of which are exponentially worse than the guarantees achievable for list update (i.e., 2 or 1.6). This gap arises because the MTS induced by list update is highly structured and far from an arbitrary metric space.

Chapter 3

Preliminaries and Problem Statement

In what follows, we present a formal description of the problem setting and the fundamental tools considered in this thesis.

3.1 Problem Statement

The uniform cost model (paid-exchanged model with $d = 1$) can be defined as follows.

Definition 3.1.1 (Uniform Cost Model [6, 9])**.** *Let $L = x_1 \to x_2 \to \ldots \to x_\ell$ denote a linear ordering of ℓ distinct items, which specifies the initial ordering of ℓ items in a linked list. An online algorithm receives a request sequence $\sigma = \langle r_1, r_2, \ldots, r_n \rangle$, where each request r_t refers to an item currently contained in L. The requests appear online, and upon arrival of r_t, an online algorithm is oblivious to future requests $r_{t'}$ where $t' > t$.*

Under the partial cost model, a requested item r_t at position i in the list, with positions numbered starting from 1*, is accessed by scanning the list sequentially from*

the front. The accessed item itself does not contribute to the access cost. Thus, the cost is equal to the number of items preceding it at the time of the request, i.e., cost of $access_t(r_t) = i - 1$. *In the full cost model, the accessed item is also counted, and hence cost of* $access_t(r_t) = i$.

After servicing a request, the algorithm may modify the list's ordering through paid exchanges. A paid exchange consists of swapping two adjacent items in the list, with each such transposition incurring a unit cost. If P *denotes the total number of paid exchanges by the online algorithm* A, *its total cost would be* $A(\sigma) = P + \sum_{t \in [1,n]} cost\ of\ access_t(r_t)$.

Note that $\text{ALG}_{\text{full}}(\sigma) = \text{ALG}_{\text{partial}}(\sigma) + n$, where ALG_{full} and $\text{ALG}_{\text{partial}}$ denote the costs incurred by ALG under the full and partial cost models, respectively. This follows because the access cost under the full cost model exceeds that under the partial cost model by exactly 1 for each request.

Observe that the above definition does not include (allow) free exchanges. That is, moving the accessed item anywhere closer to the front requires paid exchanges (each at a cost of 1 in the uniform cost model).

It is also worth noting that the offline problem in the uniform cost model is equivalent to the offline problem in the standard cost model. To see this, consider an offline algorithm in the standard model that uses a free exchange to move an accessed item closer to the front of the list. Since the entire request sequence is known in advance, the same rearrangement can instead be performed before the item is requested, using paid exchanges. The cost of these exchanges is exactly the cost that would have been incurred by accessing the item in its original position. Thus, the optimal offline costs in the two models are the same. Since the offline list update

problem under the standard cost model is NP-hard, the corresponding offline problem under the uniform cost model is also NP-hard.

3.2 Analysis Tools

In what follows, we will explain the competitive ratio and analysis tools, such as the potential function method and the projective property that allows easier analysis of many list update algorithms.

3.2.1 Comeptitvie Ratio

Our main criterion for analysis of online algorithms is the standard competitive ratio, defined formally as follows.

Definition 3.2.1 (Competitive Analysis for Minimization Probelms [5, 9])**.** *Let $A(\sigma)$ denote the total cost incurred by a deterministic online algorithm A on a sufficiently long request sequence σ, and let $\textsc{Opt}(\sigma)$ denote the cost incurred by an optimal offline algorithm $\textsc{Opt}$ that has unbounded computational power and complete knowledge of σ in advance.*

A deterministic online algorithm A is said to be ***c*-competitive** *if there exists a constant $\alpha \geq 0$ such that $\forall \sigma : A(\sigma) \leq c \cdot \textsc{Opt}(\sigma) + \alpha$. The* **competitive ratio** *of A is the smallest value of c for which the above inequality holds.*

For a randomized online algorithm A against an oblivious adversary, the notion of competitiveness is defined analogously to the deterministic case. The only difference is that the cost of the online algorithm is replaced by its expected cost, taken over the internal randomness of the algorithm. That is, $A(\sigma)$ is replaced by $\mathbb{E}[A(\sigma)]$,

while the offline optimum $\textsc{Opt}(\sigma)$ *remains unchanged. That is, A is c-competitive iff* $\forall\sigma,\ \mathbb{E}[A(\sigma)] \leq c \cdot \textsc{Opt}(\sigma) + \alpha$.

For non-trivial inputs, the asymptotic performance of an online algorithm is effectively determined by the multiplicative factor c. Equivalently, for sequences for which the cost of Opt grows asymptotically with input length, the quantity of interest is the ratio $\frac{A(\sigma)}{\mathrm{OPT}(\sigma)}$, while α contributes only lower-order terms that are independent of σ.

3.2.2 Potential Function Method

A common technique for analyzing online algorithms is the ***potential function method***, a fundamental approach to the ***amortized analysis*** of algorithms and data structures. The main idea is to associate each ***configuration*** of the online algorithm and the optimal offline algorithm with a nonnegative real-valued function, called the ***potential function***. The potential function can be viewed as a measure of "stored work" that can be used to account for future costs or compensate for costs incurred by previous operations. We then define the ***amortized cost*** of each operation by combining its actual cost with the change in potential. By summing these amortized costs over the entire request sequence, we can bound the average cost per operation, rather than analyzing each operation individually.

Let A be an online algorithm and let Opt denote an optimal offline algorithm. For each time step t, let A_t and $\textsc{Opt}_t$ denote their respective configurations after serving the first t requests. In the context of the list update problem, an algorithm's configuration is the list state it maintains. A potential function is a mapping $\Phi : (A_t, \textsc{Opt}_t) \to \mathbb{R}_{\geq 0}$.

Suppose that serving request r_t incurs costs $c_A(t)$ and $c_{\text{OPT}}(t)$ for A and OPT, respectively. Define the change in potential by

$$\Delta\Phi_t = \Phi(A_t, \text{OPT}_t) - \Phi(A_{t-1}, \text{OPT}_{t-1}).$$

To prove that A is c-competitive, it suffices to show that for every request r_t: $c_A(t) + \Delta\Phi_t \leq c \cdot c_{\text{OPT}}(t)$. The $c_A(t) + \Delta\Phi_t$ term is referred to as the amortized cost of serving request r_t. The reason this is sufficient to establish c-competitiveness is that, when we sum the amortized costs over all requests in a sequence σ, the changes in the potential function telescope, allowing us to relate the total amortized cost to the total actual cost of the algorithm:

$$\begin{aligned}
&c_A(1) + \Delta\Phi_1 \;+\; c_A(2) + \Delta\Phi_2 \;\cdots\; +\; c_A(f) + \Delta\Phi_f \\
&= c_A(1) + c_A(2) + \cdots + c_A(f) + \Phi_f - \Phi_0 \\
&= \boldsymbol{A(\sigma) + \Phi_f - \Phi_0} \\
&\leq c \cdot c_{\text{OPT}}(1) \;+\; c \cdot c_{\text{OPT}}(2) \;+\; \cdots + \; c \cdot c_{\text{OPT}}(f) \\
&= \boldsymbol{c \cdot \text{OPT}(\sigma)},
\end{aligned}$$

where Φ_0 and Φ_f denote the initial and final potential values, respectively. Rearranging gives $A(\sigma) \leq c \cdot \text{OPT}(\sigma) + \Phi_0 - \Phi_f$. and if the potential function is nonnegative and the initial potential is bounded by a constant α independent of input length, then: $A(\sigma) \leq c \cdot \text{OPT}(\sigma) + \alpha$, which establishes that A is c-competitive.

3.2.3 Projective Property

A fundamental concept in the analysis of list update algorithms is the ***projective property***. An online list update algorithm is said to be ***projective*** if the relative order of any two items depends only on the requests made to those two items and is independent of requests to all other items.

Formally, for any pair of distinct items $x, y \in L$ and any request sequence σ, consider the projection of σ onto $\{x, y\}$, denoted by σ_{xy}, obtained by removing all requests to items other than x and y. An algorithm is projective if the relative order of x and y after processing σ is completely determined by their initial relative order and the projected sequence σ_{xy}. When the projective property holds, the total cost of an online list update algorithm can be decomposed into pairwise contributions between items.

For a request sequence σ, the access cost of an item is its position in the list, i.e., the number of elements preceding it. Thus, each access cost for a request to item x can be viewed as being distributed over all pairs (y, x) such that y precedes x at the time of access. Hence, the total cost of a projective algorithm A under the partial cost model (where access to a request at position i has cost $i - 1$) can be written as

$$A(\sigma) = \sum_{(x,y)} A_{xy}(\sigma_{xy}),$$

where A_{xy} is algorithm A which maintains a list containing only x and y.

This decomposition of cost over projections is valid because the algorithm's projective property ensures that the behavior of each pair is **independent** of all other items.

As an example, the MTF algorithm is projective. To see why, consider any two items x and y. Their relative order after serving a request depends only on which of the two items was requested most recently: the more recently requested item is moved to the front, and therefore appears before the other. Requests to any other items do not affect the relative order of x and y. Thus, the relative ordering of every pair of items can be determined solely by the requests involving that pair, which is precisely the projectivity property.

On the other hand, the TRANSPOSE algorithm, which swaps the accessed item with the item immediately preceding it, is not projective. In this case, the relative order of two items can be affected by requests to other items. For example, an item may move forward in the list as a result of being accessed, but whether it eventually passes another item depends on the requests made to items between them. Thus, the relative ordering of a pair of items cannot, in general, be determined solely by the requests involving that pair, and TRANSPOSE does not satisfy the projectivity property.

The projective property significantly simplifies the analysis of list update algorithms because the behavior of the algorithm can be studied through pairwise interactions between items. Most classical algorithms, including MTF, TS, and FREQUENCYCOUNT, satisfy the projective property [4]. The most useful consequence of projectivity is the following lemma:

Lemma 3.2.2. *Let A be a projective list update algorithm. To show that A is c-competitive under the partial cost model, it suffices to prove that A is c-competitive for lists of size two under the partial cost model.*

Proof. By the projective property, the behavior of A on any subset of two items

is identical to its behavior when restricted to a list containing only those two items. Therefore, the total cost of A on a list of size ℓ under the partial cost model can be decomposed into the sum of its costs over all $\binom{\ell}{2}$ pairs of items.

If A is c-competitive on every two-item list, then for every pair of items, the cost incurred by A is at most c times the cost of the optimal offline algorithm restricted to that pair. Summing over all pairs gives that the total cost of A is at most c times the sum of the pairwise optimal costs.

Since the optimal offline cost on the original list is at least the sum of the optimal costs of the pairwise projections (up to the same decomposition used for the algorithm), the total cost of A is at most c times the optimal offline cost. Hence, A is c-competitive. □

However, a competitive ratio of c obtained using the approach in the above lemma does not necessarily imply that the bound is tight. The pairwise analysis compares the algorithm against optimal offline solutions on two-item projections, but these pairwise optimal solutions may not correspond to a single optimal offline strategy on the original list. Therefore, the sum of the pairwise optimal costs may be strictly smaller than the true optimal offline cost, leaving a possible gap between the proven bound and the actual competitive ratio.

Combining Lemma 3.2.2 with the following lemma, we can use projection to find upper bounds for the full cost model as well:

Lemma 3.2.3. *Any upper bound on the competitive ratio of an algorithm in the partial cost model also holds in the full cost model.*

Proof. This is because, for any algorithm A, for the cost of A for serving an input σ of length n under full and partial cost models, we can write $A_{full}(\sigma) = A_{partial}(\sigma) + n$.

Therefore, for the competitive ratio of an online algorithm ALG under the two models, we can write:

$$\frac{\text{ALG}_{partial}(\sigma)}{\text{OPT}_{partial}(\sigma)} \leq c \Rightarrow \frac{\text{ALG}_{partial}(\sigma)+n}{\text{OPT}_{partial}(\sigma)+n} \leq c \Rightarrow \frac{\text{ALG}_{full}(\sigma)}{\text{OPT}_{full}(\sigma)} \leq c$$

□

As an additional simplification, an optimal offline algorithm for lists of size two is known:

Definition 3.2.4 (Offline Algorithm OPT2)**.** *Write the input sequence in the form $x^{a_1}y^{a_2}x^{a_3}...y^{a_q}$, where each a_i is a positive (nonzero) integer. At the beginning of any block of form x^{a_i} (and similarly y^{a_i}) OPT2 moves x (y) to the front if and only if $a_i \geq 2$.*

The optimality of OPT2 under the standard model was established by Reingold and Westbrook [16]. The same proof applies to the uniform cost model.

Theorem 3.2.5. *[16] OPT2 is an optimal offline algorithm for lists of size 2 under the uniform cost model.*

Chapter 4

Randomized Lower Bound

4.1 Overview

In this section, we establish a lower bound of 2.5 (Theorem 4.3.2) for the competitive ratio of any randomized algorithm for uniform list update with partial costs. Recall that we consider a standard adversarial model which assumes an ***oblivious adversary*** who knows how the algorithm works (its code) but is oblivious to its random choices (random bits used by the algorithm). To the best of our knowledge, the best existing lower bound is 1.5, presented in the work of Jenke and Albers [1].

Yao's Min-Max Theorem. Our lower bound works even for lists of size 2. In particular, we present a distribution for the input sequence and assume a ***deterministic*** algorithm A is serving such ***stochastic input***. We show that the expected cost of the deterministic algorithm is no less than 2.5 times that of an optimal solution. Applying Yao's min-max theorem [39], we may conclude that the competitive ratio of any ***randomized algorithm*** R against an adversarial sequence is also at least 2.5.

In particular, Yao's min-max theorem states that

$$\min_{\text{randomized } R} \max_{\sigma} \frac{\mathbb{E}\left[R(\sigma)\right]}{\textsc{Opt}(\sigma)} \;\geq\; \max_{\mu} \min_{\text{deterministic } A} \frac{\mathbb{E}_{\sigma\sim\mu}[A(\sigma)]}{\mathbb{E}_{\sigma\sim\mu}[\textsc{Opt}(\sigma)]},$$

where μ ranges over distributions on input sequences.

Thus, to prove a lower bound of 2.5 on the competitive ratio of randomized algorithms, it suffices to present an input distribution μ for which every deterministic algorithm has an expected cost that is at least 2.5 times the expected cost of Opt.

In the remainder of this section, we will assume the input is stochastic and served by a deterministic online algorithm.

We first prove a structural lemma in Section 4.2 and present our main result (the input distribution and cost analysis) in Section 4.3.

4.2 A Structural Lemma

In this section, we prove a structural lemma that establishes a lower bound for the expected cost of an algorithm against a subsequence of inputs. We assume the partial cost model on lists of length 2, formed by items x and y.

Lemma 4.2.1. *Assume the current list order is $x \to y$, and before the first request arrives, the algorithm may swap x and y using a paid exchange. Suppose the input continues with $xxxx$ with probability $1/4$ and with $yyyy$ with probability $3/4$. Then every deterministic online algorithm for the uniform cost model (partial costs) incurs an expected cost of at least 1.5 for these five requests.*

Proof. Initially, the list is ordered as $x \to y$. Since the upcoming subsequence

consists entirely of either x's or y's, a ***smart*** algorithm has only two meaningful choices before the first request arrives, reflecting a 'guess' on the type of subsequence:

- ***wait-and-confirm***, leaving the list unchanged until the first request reveals the subsequence type, thereby guessing that the subsequence consists of x's, or
- ***preemptively-move*** y to the front, thereby guessing that the subsequence consists of y's.

Any additional exchanges would only increase the cost and are therefore unnecessary. We show that both strategies have the same expected cost of 1.5. Therefore, regardless of which strategy a smart algorithm adopts, its behavior can be viewed as a convex combination of the two options, each with expected cost 1.5. By a smart algorithm, we mean one that knows the distribution of the input sequence and can adapt its strategy accordingly. While online algorithms typically have no knowledge of future requests, allowing this additional information gives the algorithms more power and therefore strengthens our results. The details of computing the costs for both strategies are provided below and summarized in Table 4.1.

- Suppose the continuation is $xxxxx$ (which occurs with probability $1/4$). There are two cases to consider:
  - If the algorithm chooses to wait-and-confirm, then x is already at the front, and none of the requests incur any cost. So the total cost is 0.
  - On the other hand, if the algorithm chooses to preemptively-move y before x (which is indeed a mistake), it pays 1 for the initial exchange, 1 for accessing the first x, and at least 1 for the remaining four requests (the

	probability	wait-and-confirm	preemptively-move
$xxxxx$	1/4	0	3
$yyyyy$	3/4	2	1

Table 4.1: Sequence probabilities and costs for Lemma 4.2.1 (assuming initial ordering of $x \to y$).

best option is to pay 1 and move x back to the front for the remaining requests). The total cost is therefore at least 3.

- Suppose the continuation is $yyyyy$ (which occurs with probability 3/4). As before, there are two cases to consider;
  - If the algorithm chooses to wait-and-confirm (keeping x in the front), then after the arrival of the first y, it realizes its mistake. The first request to y costs 1, after which the algorithm incurs at least an additional 1 for the remaining requests (its best option is to move y to the front at an additional cost of 1, with the remaining requests free). Therefore, the total cost of the algorithm is at least 2.
  - If the algorithm chooses to preemptively-move y before x, then it incurs no access cost during the sequence. Its total cost is therefore 1, corresponding to the initial preemptive exchange.

To summarize, if the algorithm chooses to wait-and-confirm, its expected cost is $\frac{1}{4} \cdot 0 + \frac{3}{4} \cdot 2 = 1.5$. On the other hand, if it chooses to preemptively-move y before x, its expected cost is $\frac{1}{4} \cdot 3 + \frac{3}{4} \cdot 1 = 1.5$

Therefore, every randomized online algorithm incurs an expected cost of at least 1.5 (including the first exchange before the input arrives), completing the proof. □

4.3 The Main Result

Before proving our main result, we make the following observation.

Observation 4.3.1. *To serve the request sequence $xxxxx$, regardless of the initial configuration of the list or the desired final ordering, it is optimal to move x to the front of the list immediately and keep it there. In particular, even if it is desired to end with the ordering $y \to x$, there is an optimal strategy that moves x to the back only* **after** *serving the final request.*

This observation holds because having many consecutive requests for the same item makes it necessary to keep that item at the front of the list in any optimal solution. Now, we are ready to state and prove the main result of this chapter.

Theorem 4.3.2. *Any randomized online algorithm for the uniform cost model (with partial costs) has a competitive ratio of at least* 2.5.

Proof. Construct the request sequence as a series of blocks over the two items x and y. W.l.o.g., assume that the previous block ends with x (for the first block, assume that x is the item initially at the front of the list). If instead the previous block ends with y, simply swap the roles of x and y and construct the next block symmetrically. The next block is then generated according to the following probability distribution (see also Table 4.2):

- Block $x\,xxxxx$ with a chance of 1/3,
- Block $y\,xxxxx$ with a chance of 1/6,
- Block $y\,yyyyy$ with a chance of 3/6.

Recall that the previous block ended with x. We first show that any smart algorithm has the list in order $x \to y$ immediately after accessing the last request of the previous block (before any paid exchanges are performed). This is because, after observing the second request of the previous block, a smart algorithm can determine the remainder of the block (either all x's or all y's). By Observation 4.3.1, we may assume that whenever a block ends with a request to x, the algorithm's list is ordered as $x \to y$ immediately before any paid exchanges for the final request. If the algorithm chooses to reorder the list to $y \to x$ before the next block begins, this cost will be charged to the analysis of the subsequent block.

Before the first request of the next block arrives, similar to Lemma 4.2.1, the algorithm has two possible choices: it may either preemptively-move y to the front of the list, or leave the ordering unchanged. The analysis of these three blocks is summarized in Table 4.2 and further explained below.

- Suppose the next block is $x\,xxxxx$ (which occurs with probability 1/3).
  - If the algorithm chooses to wait-and-confirm, then after the arrival of the first x, it can deduce that the remainder of the block consists entirely of x's, and it has been lucky in this case, incurring zero cost for the entire block.
  - However, if it chooses to preemptively move y to the front at a cost of 1, it realizes its mistake once the first x is revealed. It then incurs an additional access cost of 1 and a paid exchange cost of 1 to move y back, for a total cost of at least 3.
- Suppose the next block is $y\,xxxxx$ (which occurs with probability 1/6).

- Assume the algorithm chooses to wait-and-confirm and does not change the list order initially. After the first request for y, it pays an access cost of 1 and learns that the remainder of the block is $xxxxx$ with probability $1/4$ and $yyyyy$ with probability $3/4$. Since the algorithm has not changed the order, the list remains $x \to y$. By Lemma 4.2.1, the expected cost for the remaining requests in the block is at least 1.5, giving a total expected cost of at least 2.5 for the whole block.
- If the algorithm instead chooses to preemptively-move, it incurs a cost of at least 1 due to the initial paid exchange.

- Suppose the next block is $y\,yyyyy$ (which occurs with probability $3/6$). The analysis is analogous to that of $y\,xxxxx$. The algorithm incurs a cost of 1 for the first y, regardless of whether it chooses to wait-and-confirm or preemptively move. By Lemma 4.2.1, the expected cost for the remaining requests in the block is at least 1.5. Thus, the total expected cost for the block is at least 2.5.

	probability	wait-and-confirm	preemptively-move	OPT
$x\,xxxxx$	$1/3$	0	3	0
$y\,xxxxx$	$1/6$	$1 + 1.5$	≥ 1	1
$y\,yyyyy$	$3/6$	$1 + 1.5$	≥ 1	1

Table 4.2: Block probabilities and costs for Theorem 4.3.2 (assuming initial ordering of $x \to y$).

Therefore, the expected cost of preemptively-move is at least $\frac{1}{3}\cdot 0+\frac{1}{6}\cdot 2.5+\frac{3}{6}\cdot 2.5 = \frac{5}{3}$, and the expected cost of wait-and-confirm is at least $\frac{1}{3}\cdot 3+\frac{1}{6}\cdot 1+\frac{3}{6}\cdot 1 = \frac{5}{3}$. And regardless of which strategy a smart algorithm adopts, its behavior can be viewed as a convex combination of the two options. Hence, the expected cost per block is at least $5/3$.

Meanwhile, the cost of Opt for each block can be tracked by the behavior of Opt2, the optimal offline algorithm for lists of size 2, defined in Definition 3.2.4. In particular, Opt2 moves an item to the front only when it is requested at least twice consecutively. Thus, it can be viewed as a deterministic algorithm that always makes the correct decision before the next block arrives: either to preemptively-move or to wait-and-confirm. Thus, the expected cost of Opt per block (as shown in Table 4.2) is $\frac{1}{3}\cdot 0+\frac{1}{6}\cdot 1+\frac{3}{6}\cdot 1 = \frac{2}{3}$.

Therefore, by Yao's min-max theorem, the competitive ratio of any randomized algorithm is at least $\frac{5/3}{2/3} = 2.5$, completing the proof. □

Chapter 5

A Randomized Algorithm

In this chapter, we will present an algorithm AltComb which achieves a competitive ratio of $34/13 \lessapprox 2.616$, improving over the best existing algorithm of [2], which has a competitive ratio of $\sqrt{7} \approx 2.646$. We first review, in Section 5.1, some classic algorithms for this model and explain how they work and their competitive ratios. We present our algorithm in Section 5.2 and analyze it in Section 5.3.

5.1 Overview

MTF is 4-competitive in the uniform cost model. To see this, consider an adversary that repeatedly requests the last item in the list. For each request, MTF incurs an access cost of ℓ and then performs ℓ paid exchanges to move the accessed item to the front, resulting in a total cost of 2ℓ per request. In contrast, if Opt leaves the list unchanged, its access costs follow the repeating sequence $(\ell, \ell - 1, \ldots, 1)$. The average asymptotic access cost per request is therefore $\ell/2$ for large ℓ. Hence, the competitive ratio is at least $\frac{2\ell}{\ell/2} = 4$. Therefore, MTF is at least 4-competitive. This

bound is trivially tight, as moving each item to the front doubles the cost of MTF in the uniform cost model compared to the standard model.

As with MTF, TS is no better than 4-competitive in the uniform cost model. Consider an adversary that cyclically requests every item in the list, starting with the last item. After the first cycle, TS behaves identically to MTF, moving each requested item to the front and incurring a cost of 2ℓ per request. As in the analysis of MTF, an optimal offline algorithm can leave the list unchanged, achieving an average access cost of asymptotically $\ell/2$ per request. Hence, TS has an asymptotic competitive ratio of at least 4.

Recall that in the uniform cost model, the lower bound on the competitive ratio of deterministic algorithms is 3. Therefore, unlike in the standard cost model, TS and MTF are not optimal. The reason these algorithms are optimal in the standard cost model but not in the uniform cost model is that they rely heavily on free exchanges and consequently move items too aggressively. In the uniform cost model, where every exchange incurs a cost, effective algorithms must be more conservative in performing list updates.

Basiak et al. [6] addressed this issue via a deterministic algorithm Full-Or-Partial-Move (FPM), which, as the name suggests, reduces the frequency of full-moves-to-the-front in order to find a better balance. They prove that this algorithm has a competitive ratio of at most 3.3904. However, the resulting analysis is quite involved. With the use of randomization, there are more straightforward approaches to this problem:

A natural first candidate is RMTF (as described in Section 2.1), which moves the requested item to the front with probability 0.5. However, using the same request

sequence as in Section 2.1, it is easy to show that RMTF is at least 3-competitive in the uniform cost model. Since the lower bound for randomized algorithms is 2.5, this leaves a gap. One possible approach is to generalize RMTF by moving the requested item to the front with probability p. However, this introduces a trade-off: when $p > 0.5$, the algorithm becomes more similar to MTF and performs poorly on cyclic requests, where items are repeatedly requested in order. On the other hand, when $p < 0.5$, the algorithm becomes less aggressive and struggles on sequences where individual items are requested many times consecutively.

Another natural approach is to avoid making a random decision on every request and instead randomize only at the beginning to confuse the adversary's choice of strategy. This is the idea behind the COUNTER family, which achieves a competitive ratio of $\sqrt{7} \approx 2.646$ [2] and is currently the best-known algorithm for the uniform cost model.

BIT is a simple member of the COUNTER family, which we later in Section 5.3 show to be 2.75-competitive. A particular type of input (over two items) on which BIT performs poorly over is $(xyxx)^k$ (later formalized as a Type 3 phase with $i = 1$). However, TS performs well on these inputs.

This motivates combining the two algorithms to obtain improved performance. In the standard model, Alberse [11] introduced the COMB algorithm, which randomly combines TS and BIT for improved competitive ratios: at the beginning, it selects BIT with a chance of 4/5 and TS with a chance of 1/5 and applies the selected algorithm to serve the entire input. In the following section, we also combine the same two algorithms to introduce ALTCOMB, although with a different approach.

5.2 Algorithm AltComb

To review, we briefly restate the definitions of Bit and TimeStamp (TS):

- The algorithm Bit operates as follows: Initially, each item x in the list is assigned an independent uniformly random bit $b \in \{0, 1\}$, denoted by $\overset{b}{x}$. Upon each request to an item, its bit is flipped. If the bit is 0 before the access, no further action is taken. If the bit is 1 before the flip (and therefore changes to 0), the item is moved to the front of the list.

- The algorithm TS works as follows: After an item x is accessed, TS moves x forward in the list until it is positioned immediately before the first item y that precedes x and has been requested at most once since the previous access to x. If no such item y exists, or if x has never been accessed before, TS leaves the list unchanged and performs no reordering.

We now describe the algorithm AltComb:

Definition 5.2.1 (Algorithm AltComb)**.** *Define the randomized algorithm AltComb as follows. At the beginning of the execution, AltComb selects algorithm Bit with probability* $12/13$ *and algorithm TS with probability* $1/13$*, and then follows the selected algorithm for the entire request sequence.*

Note that AltComb is a barely random algorithm, meaning that the number of used random bits is independent of the input length (it is $O(1)$ with respect to the input length); all randomness is used during initialization, and no additional random bits are required afterward. Specifically, it first uses $\lceil \log(13) \rceil = 4$ bits to choose which algorithm to use. If it selects Bit, an additional ℓ random bits are needed

to initialize the bit values for each item, while TS requires no further randomness. Therefore, the total number of random bits is at most $\ell + 4$. Compared to other algorithms, such as FPM, which is currently the best-known deterministic algorithm and relies on many rules and calculations to determine the appropriate move after each request, AltComb is considerably simpler.

5.3 Analysis of AltComb

We prove the following upper bound for the competitive ratio of AltComb.

Theorem 5.3.1. *The algorithm AltComb is* $\frac{34}{13} \approx 2.616$*-competitive for list update with uniform costs (under both partial and full cost models).*

Proof. It suffices to prove the theorem for the partial cost model, since the corresponding upper bound for the full cost model then follows from Lemma 3.2.3. Hence, throughout the remainder of the proof, we consider only the partial cost model.

Both Bit and TS are projective [4], and since AltComb essentially works identically to one of them, we conclude that AltComb is also projective. Therefore, by Lemma 3.2.2, it suffices to show that AltComb is $\frac{34}{13}$-competitive on lists of size 2. To do this, we compare the cost of AltComb with that of Opt2, an optimal offline algorithm for lists of size 2 described in Definition 3.2.4.

For every input sequence σ, we partition the sequence deterministically into phases using a technique similar to that introduced in the analysis of TS [7]. We construct the partitioning so that every phase ends with an element that appears exactly twice in a row. Consequently, at the end of each phase, all algorithms (including Opt2) would bring the same item to the front of their lists.

Assume that the first item in the initial list is x. We define the next phase as an ***x-phase*** (with ***y-phase*** defined symmetrically when y is the first item in the current list). Partition the whole input sequence as such to x and y-phases. An x-phase can be any of the following types:

- Type 1: $x^j yy$
- Type 2: $x^j (yx)^i yy$
- Type 3: $x^j (yx)^i x$

Where $j \geq 0,\ i \geq 1$. Note that this partitioning is unique; there is no alternative way to decompose the sequence into phases of these forms. Also, observe that all algorithms incur no cost on the x^j portion, since x is at the front of all algorithms' lists at the start of an x-phase.

	Bit	TS	AltComb	Opt2
$x^j yy$	2.5	3	$\frac{33}{13}$	1
$x^j (yx)^i yy$	$2.5i + 1.5$	$4i - 1$	$\frac{34}{13}\, i + \frac{17}{13}$	$i + 1$
$x^j (yx)^i x$	$2.5i + 0.25$	$4i - 3$	$\frac{34}{13}\, i$	i

Table 5.1: Costs of algorithms over each phase.

We now analyze the expected costs of Bit and TS over each phase to show that the expected cost of AltComb, given by $\frac{12}{13}\mathbb{E}[\text{Bit}] + \frac{1}{13}\mathbb{E}[\text{TS}]$, is at most $\frac{34}{13}$Opt2, completing the proof. A summary of the costs for each partition is provided in Table 5.1, with a more detailed analysis of the costs of Bit and TS given below.

- **Phases of Type 1 ($x^j yy$):** As mentioned before, since x is at the front of all algorithms, the x^j part incurs no cost.

  For **Bit**, the first request to y has expected cost $1+\frac{1}{2}$, accounting for the access cost and the possible move-to-front cost. For the second consecutive request to y, we note that y is already at the front with probability $\frac{1}{2}$, and otherwise Bit incurs an access and move-to-front cost of $1+1$, giving an expected cost of $\frac{1}{2}\cdot 0+\frac{1}{2}\cdot(1+1)$. Therefore, the total expected cost for Bit is $\left(1+\frac{1}{2}\right)+\left(\frac{1}{2}\cdot 0+\frac{1}{2}\cdot(1+1)\right)=2.5$.

  For **TS**, the first request to y only incurs an access cost of 1, since x has been requested at least twice since the previous request to y. This follows from the definition of an x-phase: for a new x-phase to begin, the preceding phase must end with exactly two consecutive requests to x. However, on the second consecutive request to y, TS accesses y at a cost of 1, and then moves it to the front of the list, resulting in a total cost of 3 for TS.

  Opt2 incurs a total cost of 1, since it moves y to the front before the first request to y (by the definition of Opt2).

  It is easy to see that the expected cost of **AltComb** is at most $\frac{34}{13}$ times the cost of Opt2. Let AltComb($x^j yy$) denote the expected cost of AltComb for this phase (we use the same notation convention for the other algorithms, in

particular Bit, TS, and Opt2). Then we can write:

$$\begin{aligned}\textsc{AltComb}(x^j yy) &= \frac{12}{13}\,\textsc{Bit}(x^j yy) + \frac{1}{13}\,\textsc{TS}(x^j yy)\\ &= \frac{12}{13}\,2.5 + \frac{1}{13}\,3\\ &= \frac{33}{13}\\ &< \frac{34}{13}\,\textsc{Opt2}(x^j yy)\end{aligned}$$

- **Phases of Type 2 ($x^j(yx)^i yy$):** Again, the x^j portion incurs no cost for any of the algorithms.

  We next show the cost of $(yx)^i$ for **Bit**. Initially, x appears before y, and the bits of both items are independently set to either 0 or 1. Therefore, when we are processing this phase, there are four possible configurations (as given in Table 5.2), each occurring with equal probability. We first show that after every $(yx)^2$, each initial configuration returns to a repeating state.

  - Consider the initial list configuration of $\overset{0}{x} \to \overset{0}{y}$. When the first request of y arrives, Bit incurs an access cost of 1. Since its bit is 0, it is not moved to the front, and no exchange cost is incurred. The following request to x incurs no cost, as x is already at the front of the list. Thus, the total cost of the first yx request is $1+0=1$. After handling the first yx requests, the resulting configuration is $\overset{1}{x} \to \overset{1}{y}$. The next request to y incurs an access cost of 1 and, since its bit is 1, y is moved to the front at an additional cost of 1, for a total cost of 2. The subsequent request to x now finds x at the back of the list, and since its bit is also 1, it is moved to the front, again

incurring a total cost of 2. Therefore, the total cost for this yx request is $2 + 2 = 4$, and the resulting list configuration is $\overset{0}{x} \to \overset{0}{y}$, which is the same configuration we started with. Therefore, after every $(yx)^2$, we end up with the starting configuration.

- Similarly, the configurations $\overset{1}{x} \to \overset{1}{y}$ and $\overset{0}{x} \to \overset{1}{y}$ also return to their initial configurations after every $(yx)^2$. The details are similar to the previous case and can be easily verified through Table 5.2.
- The configuration $\overset{1}{x} \to \overset{0}{y}$ behaves slightly differently. After the first $(yx)^2$, it returns to $\overset{0}{y} \to \overset{1}{x}$ instead of the initial configuration. But similar to the other cases, from now on, after every $(yx)^2$, it would reset to this new $\overset{0}{y} \to \overset{1}{x}$ state.

input:	yx	yx	yx	yx	yx
Configurations of BIT:	$\overset{0}{x} \to \overset{0}{y}$ (cost 1+0) $\overset{1}{x} \to \overset{1}{y}$ (cost 2+2) $\overset{0}{x} \to \overset{1}{y}$ (cost 2+1) $\overset{1}{x} \to \overset{0}{y}$ (cost 1+0)	$\overset{1}{x} \to \overset{1}{y}$ (cost 2+2) $\overset{0}{x} \to \overset{0}{y}$ (cost 1+0) $\overset{0}{y} \to \overset{1}{x}$ (cost 0+2) $\overset{0}{x} \to \overset{1}{y}$ (cost 2+1)	$\overset{0}{x} \to \overset{0}{y}$ $\overset{1}{x} \to \overset{1}{y}$ $\overset{0}{x} \to \overset{1}{y}$ $\overset{0}{y} \to \overset{1}{x}$	$\overset{1}{x} \to \overset{1}{y}$ $\overset{0}{x} \to \overset{0}{y}$ $\overset{0}{y} \to \overset{1}{x}$ $\overset{0}{x} \to \overset{1}{y}$	...
Total cost of all possible cases:	$1 + 4 + 3 + 1 = 9$	$4 + 1 + 2 + 3 = 10$	10	10	10
Expected cost:	9/4	10/4	10/4	10/4	10/4

Table 5.2: BIT's list configurations and costs over the sequence $(yx)^i$ with initial list ordering $x \to y$. After the first yx request, the configurations repeat every two requests.

Now that we have shown that the configurations repeat every $(yx)^2$, we move on to analyzing the expected cost of BIT over the sequence $(yx)^i$.

For the configuration $\overset{0}{x} \to \overset{0}{y}$, the cost of BIT over the first yx is 1. As Table 5.2 illustrates, the cost of BIT for conifigurations $\overset{1}{x} \to \overset{1}{y}$, $\overset{0}{x} \to \overset{1}{y}$, and $\overset{1}{x} \to \overset{0}{y}$ are resppectively $4, 3$, and 1. We conclude that the exepcted cost of BIT for the first yx is $(1+4+3+1)/4 = 9/4$. Repeating the same analysis for the next yx request yields an expected cost of $10/4$ for BIT. After this second yx, all configurations return to their initial state, except for configuration $\overset{1}{x} \to \overset{0}{y}$, which resets to $\overset{0}{y} \to \overset{1}{x}$ instead. Since the configurations reset after every $(yx)^2$, we only need to analyze the costs of the first few yx requests. As shown in Table 5.2, only the first yx request differs, with an expected cost of $9/4$, and the expected cost will be $10/4$ for every subsequent yx request.

Therefore, the expected cost over the sequence $(yx)^i$ for BIT is $\frac{9}{4} + (i-1)\frac{10}{4} = 2.5i - 0.25$ (since $i \geq 1$ by phase defenition, this process is well-defined). Looking at Table 5.2, after the $(yx)^i$ portion, the probability that y is at the back of the list (no matter if i is even or odd) is $\frac{3}{4}$ (and in two of these three cases, it will be moved to the front upon a request to y). After a request to y, the probability that y remains at the back is $\frac{1}{4}$. Therefore, the expected cost for serving the two consecutive requests to y is $\frac{3}{4} \cdot (1 + \frac{2}{3} \cdot 1) + \frac{1}{4} \cdot (1+1) = 1.75$. In total, BIT has an expected cost of $2.5i - 0.25 + 1.75 = 2.5i + 1.5$ to serve the entire phase.

To find the expected cost of **TS** for the phase, consider the $(yx)^i$ portion. The first request to y in the first yx pair incurs an access cost of 1, but no move is made (by an argument similar to the one used when analyzing TS over a Type 1 phase). The subsequent request to x incurs no cost, since x is already at the front of the list. After this first yx pair, every request to either y or x requires a paid exchange and TS moves the requested item to the front, since under TS,

an item is moved ahead of every item that has been requested at most once since its previous access. Therefore, each subsequent request to x or y incurs an access cost of 1 and a move cost of 1. Hence, the total cost of the $(yx)^i$ portion is $1+(i-1)\cdot(2+2) = 4i-3$. Again, note that by the phase definitions, it holds that $i \geq 1$, so the expression is well defined. The next request to y similarly incurs a cost of 2, while the final request to y in the phase incurs no cost since y is already at the front of the list. Therefore, the total cost for TS for the phase is $4i - 3 + 2 = 4i - 1$.

Next, we investigate the cost of **Opt2** for the phase. Opt2 incurs a cost of 1 for each yx pair, since it does not perform any moves and only pays the access cost for y. For the final two consecutive requests to y, it incurs a cost of 1 by moving y to the front of the list before the first request to y. Therefore, the total cost for **Opt2** is $i + 1$.

Again, we can show the expected cost of **AltComb** is at most $\frac{34}{13}$ times the cost of Opt2:

$$\begin{aligned}
\textsc{AltComb}(x^j(yx)^iyy) &= \frac{12}{13}\ \textsc{Bit}(x^j(yx)^iyy) + \frac{1}{13}\ \textsc{TS}(x^j(yx)^iyy) \\
&= \frac{12}{13}\,(2.5i + 1.5) + \frac{1}{13}\,(4i - 1) \\
&= \frac{34}{13}\,i + \frac{17}{13} \\
&< \frac{34}{13}\,(i + 1) \\
&= \frac{34}{13}\ \textsc{Opt2}(x^j(yx)^iyy)
\end{aligned}$$

- **Phases of Type 3 $(x^j(yx)^ix)$:** Since Type 2 and Type 3 phases only differ

in their final requests, their analyses are very similar. All algorithms incur the same cost for the $(yx)^i$ portion, and we only need to analyze the final request to x:

For **Bit**, before the final request, as Table 5.2 shows, the probability that x is at the back of the list is $\frac{1}{4}$ (in which case, it would be moved to the front). Therefore, the expected cost of this final request to x is $\frac{1}{4}(1+1) = 0.5$, giving Bit a total expected cost of $2.5i - 0.25 + 0.5 = 2.5i + 0.25$ over this phase.

For **TS**, the final request to x incurs no cost since x is already at the front of the list, giving a total cost of $4i - 3$.

For **Opt2**, x is already at the front, and no paid exchanges are required, resulting in a cost of 0 for serving the final request to x and a total cost of i.

Finally, we can show the expected cost of **AltComb** is at most $\frac{34}{13}$ times the cost of Opt2:

$$\begin{aligned}
\textsc{AltComb}(x^j(yx)^ix) &= \frac{12}{13}\ \textsc{Bit}(x^j(yx)^ix) + \frac{1}{13}\ \textsc{TS}(x^j(yx)^ix) \\
&= \frac{12}{13}\,(2.5i + 0.25) + \frac{1}{13}\,(4i - 3) \\
&= \frac{34}{13}\,i \\
&\leq \frac{34}{13}\ \textsc{Opt2}(x^j(yx)^ix)
\end{aligned}$$

Therefore, for all phases, the cost of AltComb is no more than $\frac{34}{13}$ times Opt2. Given the projective property of AltComb, by Lemma 3.2.2 and summing over all phases we conclude that the cost of AltComb is no more than $\frac{34}{13}$ times Opt under the partial cost model and thus, by Lemma 3.2.3 under the full cost model as well.

□

Chapter 6

Optimal Randomized Algorithm for Short Lists

6.1 Overview

The main result of Chapter 4 (Theorem 4.3.2) establishes that no randomized algorithm can achieve a competitive ratio better than 2.5 under the partial cost model. This lower bound holds even for lists of length 2. In this chapter, we present an algorithm for lists of length $\ell \in \{2, 3, 4\}$ and prove that it achieves a competitive ratio of 2.5. Combined with the lower bound established in Theorem 4.3.2, this proves that our algorithm is optimal.

We describe the algorithm in Section 6.2 and present its analysis in Section 6.3.

6.2 Algorithm BiPolar

In our algorithm, we make use of the ***Item-Order (IO) graph***, which represents the partial ordering of the items in the list based on the requests received so far.

Definition 6.2.1 (Item-Order (IO) graph)**.** *The* **Item-Order (IO) graph** *is a directed graph that is updated after each request in the input sequence. Vertices of the IO graph correspond to the items in the list, with each directed edge representing the relative order between a pair of items at a given time (index in the input). Initially, the graph contains no edges. To update this structure online, whenever an item is requested, add a directed edge from that item to every other vertex. If an edge in the opposite direction already exists (i.e., from another vertex to the requested item), the two edges "cancel" each other out, resulting in no edge between the two.*

Example 6.2.2. *Consider a list of three items,* 1*,* 2*, and* 3*. Initially, the IO graph contains no edges. If item* 1 *is requested, two edges are added, from* 1 *to* 2 *and from* 1 *to* 3*. If item* 2 *is then requested, an edge from* 2 *to* 3 *is added, while the new edge from* 2 *to* 1 *is canceled by the edge from* 1 *to* 2*. Thus, the resulting IO graph contains only the edges* $(1, 3)$ *and* $(2, 3)$*.*

The IO graph is designed to capture the partial ordering of items induced by Opt2, the optimal offline algorithm for lists of size 2 defined in Definition 3.2.4. For any two items x and y, if x is requested twice in a row, then Opt2 would have placed x before y. We represent this relationship via a directed edge from x to y in the IO graph. Thus, an edge from x to y indicates that, under Opt2, x precedes y in the list at the current time. If no edge exists between two items, their relative order cannot be determined from the IO graph alone. Opt2, however, still has a definite ordering,

as it has access to future requests that are not captured by the IO graph.

To make the distance between list states easier to visualize, we construct the ***List State (LS) graph***, where each vertex represents a possible list ordering, and edges correspond to a single paid exchange between two elements. For example, in the case of lists of size 3, there will be $3! = 6$ possible orderings, and in our figures, we visualize this graph as a hexagon with one ordering at each vertex. The distance between any two list states is the length of the shortest path between them in the LS graph (this would be the number of inversions, i.e., the number of paid exchanges needed to convert one ordering to the other). When describing the LS graph, we use a compact notation to save space and make the ordering easier to read. We simply list the items in their order. For example, the ordering $1 \to 2 \to 3$ is represented as 123. Note that, unlike the IO graph, which is defined based on the current index of the input sequence, the LS graph is universal and input-independent.

Intuitively, our algorithm BiPolar aims for the state of the algorithm's list to be consistent with every Opt2's state. Thus, if the IO graph indicates that an item x should appear before an item y (i.e., there is a directed edge from x to y in the IO graph), then we would like the ordering of the algorithm's actual list to respect this relation. Formally, a vertex in the LS graph, which corresponds to an ordering of the list items, is said to be ***consistent*** with the IO graph if and only if, for every directed edge from an item x to an item y in the IO graph, x appears before y in the ordering.

Example 6.2.3. *Suppose $\ell = 3$ and the current configuration of the list is $1 \to 2 \to 3$. Further, assume that the IO graph contains only two edges: one from 1 to 2 and one from 1 to 3. This list configuration is consistent with the IO graph, since item 1*

appears before both 2 *and* 3 *in the list. Now, suppose the next request is to item* 3*. In the IO graph, the edge from* 1 *to* 3 *is canceled, and a new edge from* 3 *to* 2 *is added. Consequently, the current configuration* $1 \to 2 \to 3$ *is no longer consistent with the IO graph: the edge from* 3 *to* 2 *requires item* 3 *to appear before item* 2 *in the list. As we will see,* BiPolar *updates the list whenever such an inconsistency arises.*

We are now ready to define our algorithm:

Definition 6.2.4 (algorithm BiPolar)**.** *BiPolar maintains two* **poles** *on the LS graph. At the beginning, it places one pole at the initial list ordering and the other at the ordering obtained by reversing the initial list. It then flips a coin to select one of the poles to follow. From that point onward, the actual list maintained by* BiPolar *follows the ordering represented by the selected pole, while* BiPolar *maintains the other pole virtually.*

The algorithm maintains the poles so that they satisfy the following properties:

- *By the end of each request service, both poles are located at vertices of the LS graph that are consistent with the IO graph.*
- *The poles are as far apart as possible from each other (among consistent vertices) in the LS graph, maximizing the number of pairwise disagreements between their corresponding list orderings.*

When a new request arrives and changes the IO graph, BiPolar *moves the poles to two list orderings in the LS graph that are consistent with the updated IO graph and are as far apart as possible in the LS graph. If multiple pairs of destinations satisfy these conditions, the algorithm chooses the pair that minimizes the total number of edges traversed by the two poles in the LS graph. Each pole moves on a path to its new*

state such that the combined number of edges traversed by both poles is minimized. The algorithm breaks ties arbitrarily.

Note that BiPolar is a barely random algorithm. It uses only a single bit of randomness at initialization to choose which pole to follow.

Example 6.2.5. *Suppose the current IO graph contains the edges* $1 \to 2$ *and* $1 \to 3$*. The only two vertices of the LS graph consistent with this IO graph are therefore* 123 *and* 132*. Assume that the algorithm's actual list is represented by the pole at* 123*, while the other pole is at* 132*. Now suppose the next request is to item* 3*. The updated IO graph contains the edges* $1 \to 2$ *and* $3 \to 2$*, so the new consistent vertices in the LS graph are* 132 *and* 312*. Thus, to maintain consistency with the updated IO graph, the poles must move from the two vertices* $(123, 132)$ *in the LS graph to* $(132, 312)$*. To minimize the total number of edges traversed in the LS graph, the pole at* 123 *can move to* 312*, while the pole at* 132 *remains unchanged. Since the actual list follows the pole at* 123*, this corresponds to moving item* 3 *to the front of the current list, which requires two paid exchanges.*

6.3 Analysis of BiPolar

Note that BiPolar is well-defined for any instance of list update on any list size. In this section, however, we aim to establish its optimality for the short list under the uniform cost model. To analyze the competitiveness of BiPolar, we first establish the following lemma. It allows us to relate the total cost incurred by all pairwise Opt2 algorithms (Definition 3.2.4) to the IO graph, and therefore obtain a lower bound on the cost of an optimal offline algorithm for the full list.

Lemma 6.3.1. *The sum of the costs incurred by all pairwise* Opt2 *algorithms for a request in the partial cost model can be interpreted as the number of edges in the IO graph that are canceled when that request arrives.*

Proof. Since Opt2 considers only two items, it suffices to analyze the behavior on a pair x and y. The IO graph encodes their current relative state under Opt2: if there is a directed edge from x to y, this means that x has been requested at least twice since the most recent request to y. As long as x continues to be requested, Opt2 incurs no cost. When a request to y arrives, however, Opt2 incurs a cost of 1. This can happen in one of two equivalent ways: either it moves y to the front in advance (paying for the exchange and then accessing y at zero cost), or it leaves the list unchanged and pays a unit access cost to reach y. In either case, after serving y, the edge between x and y disappears from the IO graph. At this point, the two items' relationship is "reset" with respect to each other. The next request (whether to x or y) incurs no cost for Opt2, since Opt2's previous decision already accounted for their relative order. After that request, a new edge is established in the IO graph, and the same reasoning applies again. □

To relate the pairwise costs of Opt2 to the cost of an optimal offline algorithm on lists of size ℓ, we use the following observation.

Observation 6.3.2. *The cost incurred by any optimal offline algorithm on a list of size ℓ is at least the sum of the costs incurred by the optimal offline algorithms on every pair of items.*

To see why the above observation holds, consider any pair of items in the list and project the full list onto these two items. Any access or exchange performed by the

optimal algorithm on the full list induces a corresponding action on the projected two-item list, and therefore cannot cost less than the optimal cost for that pair. Summing over all $\binom{\ell}{2}$ pairs gives a lower bound on the total cost of the optimal algorithm on the full list.

We now show that BiPolar is 2.5-competitive for lists of size 2, 3 and 4.

Theorem 6.3.3. *BiPolar is* 2.5*-competitive for the uniform cost model (both with partial and full costs), on lists of size 2, 3, and 4.*

First, by Lemma 3.2.3, it suffices to prove the upper bound for the partial cost model, as the result for the full cost model follows immediately. We use the potential function method, as described in Section 3.2.2. Define the potential function Φ as the distance (i.e., the number of inversions) between the two poles of BiPolar in the LS graph. To prove the theorem, it suffices to show that, for any IO configuration, the amortized cost of serving a request is at most 2.5 times the cost incurred by an optimal offline algorithm.

To do this, we use Observation 6.3.2 to show that the cost of an optimal offline algorithm is at least the sum of all pairwise Opt2 costs. By Lemma 6.3.1, this sum is equal to the number of canceled edges in the IO graph. We then show that the amortized cost of handling a request, $\mathbb{E}[\text{access cost}] + \mathbb{E}[\text{paid exchanges}] + \Delta\Phi$, is at most 2.5 times the number of canceled edges. Combining these two bounds yields the desired 2.5 competitive ratio. In the following sections, we analyze the amortized cost separately for lists of sizes 2, 3, and 4.

Lists of Size 2

For a two-item list consisting of items 1 and 2, the LS graph consists of only two vertices, corresponding to the list orderings 12 and 21. Note that there are two possibilities for the IO graph when it contains only two items. First, there is an edge from one item to the other item. Second, there is no edge in the IO graph. We consider these two cases separately.

- **Case I: There is an edge in the IO graph.**

  Suppose, w.l.o.g., that the edge is from 1 to 2. The case where the edge is from 2 to 1 is handled symmetrically. In this situation, only one vertex of the LS graph, namely 12, is consistent with the IO graph, and thus both of BiPolar's poles must be on the 12 state in the LS graph. Therefore, BiPolar's list ordering is definitely 12. There are two possibilities for the next request:

  - **Case Ia:** If the next request is to item 1, the number of canceled edges in the IO graph is 0. Since the IO graph does not change, the poles do not need to move and since 1 is in the front, the total cost incurred by BiPolar in serving the request is 0. Also, since the poles do not move, the potential remains unchanged. Hence, the amortized cost is also 0, which is at most 2.5 times the number of canceled edges in the IO graph: $0 \leq 2.5 \cdot 0$

  - **Case Ib:** If the next request is to item 2, the edge from 1 to 2 is removed from the IO graph as a result of the request. After serving the request, the IO graph contains no edges, and therefore, both vertices in the LS graph are consistent with the updated IO graph. In particular, 21 is also consistent

now. To maximize the distance between the two poles, one of BiPolar's poles must move from 12 to the ordering 21. Since BiPolar chose which pole to follow uniformly at random at initialization, the pole that moves corresponds to the actual list with probability 1/2. The potential also increases by 1, since the distance between the two poles increases from 0 to 1.

Thus, BiPolar incurs an amortized cost consisting of the following elements: i) an access cost of 1 for serving the request, ii) an expected cost of $1/2 \cdot 1$ for paid exchange costs (since the actual list is the pole that moves with probability 1/2), and iii) an increase of 1 in potential. Therefore, the total amortized cost is $1 + \frac{1}{2} \cdot 1 + 1 = 2.5$. Which is at most 2.5 times the number of canceled edges in the IO graph: $2.5 \leq 2.5 \cdot 1$

- **Case II: There is no edge in the IO graph.** Since the IO graph contains no edges, both orderings in the LS graph are consistent with the IO graph. Therefore, to maximize the distance between the two poles, one pole of BiPolar is placed at each of the two orderings (each of the two vertices of the LS graph).

  W.l.o.g., assume the next request is to 1 (the case for 2 is symmetric). Therefore, the IO graph is updated to include an edge from 1 to 2. No edges are removed. After the request, the updated IO graph is consistent only with the ordering 12, the pole at 21 must move to the vertex corresponding to 12.

  There are two possibilities for BiPolar. It can be either in 12 or 21 before the request is served (each with a chance of 1/2). If it is in 12, it incurs an access cost of 0 and makes no paid exchanges (it remains in state 12). So, the cost in

this case is 0. On the other hand, if BiPolar is in 21 before the request, its access cost will be 1, and it further pays a cost of 1 to move from state 21 to 12, summing up to a cost of 2 for BiPolar in this case. We conclude that the expected cost of BiPolar is $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 2 = 1$. To find the amortized cost, we note that the potential has decreased by 1 because the two poles are collocated and their distance is reduced by 1 in the LS graph. So, the amortized cost of BiPolar is $1 - 1 = 0$, which is at most 2.5 times the number of canceled edges in the IO graph: $0 \leq 2.5 \cdot 0$.

We have shown that, for all IO graph configurations for lists of size 2, the amortized cost of any request is at most 2.5 times the number of canceled edges in the IO graph. Thus, by Lemma 6.3.1 and Observation 6.3.2, BiPolar is 2.5-competitive for lists of size 2.

Lists of Size 3

The LS graph for 3 items (1, 2, and 3) has $3! = 6$ distinct orderings, which can be depicted as a hexagon. There are 5 distinct IO graph configurations for a list of size 3. All possible IO graphs, along with the corresponding vertices of the LS graph that are consistent with each configuration, are shown in Figure 6.1.

In each subfigure of Figure 6.1, the IO graph is shown in the top-left corner, while the LS graph is displayed on its right. The list states in the LS graph that are consistent with the given IO graph are enclosed within the solid boundary. At each given time, BiPolar's two poles are positioned as far apart as possible within this region. The dotted boundary in the LS graph indicates the set of vertices that are consistent with the updated IO configuration after a specified request (as labeled

green on each boundary). We now continue the analysis of amortized costs for each distinct IO configuration:

- **Case I:** There is one edge in the IO graph (configuration of Figure 6.1I):
  - **Case Ia:** Suppose the next request is to item 1. After the request, the new IO graph gains a new edge from 1 to 3, but no edge is removed. Since the IO graph changed, BiPolar's poles should move from vertices $(123, 312)$ to $(123, 132)$. To minimize the total number of edges traversed by the two poles, the pole at 123 remains in place while the pole at 312 moves to 132, traversing one edge. The expected access cost of the request to item 1 is $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = 0.5$, since, when the request arrives, the two poles are at 123 and 312. The expected paid exchange cost is also $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = 0.5$, since only one pole moves by one edge. Finally, the potential decreases by 1, as the distance between the two poles decreases by 1. Thus, the amortized cost is $0.5 + 0.5 - 1 = 0$, which is at most 2.5 the number of canceled edges in the IO graph: $0 \leq 2.5 \cdot 0$.
  - **Case Ib:** Suppose the next request is to item 2. The request removes the edge from 1 to 2 in the IO graph (and adds another edge). To move the poles to their new consistent destinations while minimizing the total number of edges traversed, BiPolar moves the pole at 312 to 231, while leaving the pole at 123 unchanged. The expected access cost for a request to item 2 is $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = 1.5$, since, when the request arrives, the two poles are at 123 and 312. The expected paid exchange cost is $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 2 = 1$, since the pole at 312 moves two edges to 231. The potential does not change,

since the distance between the two poles remains the same. Therefore, the amortized cost is $1.5 + 1 + 0 = 2.5$, which is exactly 2.5 times the number of canceled edges in the IO graph: $2.5 \leq 2.5 \cdot 1$.

- **Case Ic:** Suppose the next request is to item 3. Two new edges are added to the IO graph, but no edges are removed. After the request, the only vertex consistent with the IO graph is 312, so the pole at 123 must move to 312. The expected access cost is $\frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 0 = 1$; this is because the poles are initially at 123 and 312. The expected paid exchange cost is $\frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 0 = 1$, since one pole moves two edges to reach 312. Finally, the potential decreases by 2, since the two poles end at the same vertex. Thus, the amortized cost is $1+1-2 = 0$, which satisfies the desired bound: $0 \leq 2.5 \cdot 0$.

- **Case II:** The IO graph contains two edges with the same starting point (configuration of Figure 6.1II):

  - **Case IIa:** Suppose the next request is to item 1. A request to item 1 causes no change to the IO graph. Since the IO graph does not change, the poles do not need to move. The expected access cost is $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 0 = 0$, since the poles are initially at 123 and 132, with item 1 at the front in both orderings. Neither pole incurs a paid exchange, and the potential remains unchanged. Therefore, the amortized cost is $0 + 0 + 0 = 0$, which is at most 2.5 times the number of canceled edges in the IO graph: $0 \leq 2.5 \cdot 0$.

  - **Case IIb:** Suppose the next request is to item 2. One edge is added, and one edge is removed from the IO graph. After a request to item 2, the

poles must move from 123 and 132 to 123 and 213. There are two ways to achieve this: either the pole at 132 moves directly to 213 while the pole at 123 remains fixed, or the pole at 123 moves to 213 while the pole at 132 moves to 123. In either case, the total number of edges traversed by the two poles is 2. The expected access cost is $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = 1.5$, since the poles are initially at 123 and 132. The expected paid exchange cost is $\frac{2}{2} = 1$, since the two poles traverse a total of 2 edges, and each pole is followed by BiPolar with probability $\frac{1}{2}$. The potential remains unchanged, since the distance between the two poles is still 1. Therefore, the amortized cost is $1.5 + 1 + 0 = 2.5$, which exactly matches the desired bound: $2.5 \leq 2.5 \cdot 1$.

  - **Case IIc:** Suppose the next request is to item 3. The analysis for a request to item 3 is symmetric to a request to item 2 (Case IIb).

- **Case III:** The IO graph contains two edges with the same ending point (configuration of Figure 6.1III):

  - **Case IIIa:** Suppose the next request is to item 1. One edge is added to the IO graph, but no edge is removed. After the request, both poles must be at 132, so the pole at 312 must move to 132. The expected access cost is $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = 0.5$, since the poles are initially at 132 and 312. The expected paid exchange cost is $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = 0.5$, since one pole moves one edge to reach 132. The potential decreases by 1, since the two poles end at the same vertex. Therefore, the amortized cost is $0.5 + 0.5 - 1 = 0$, which is at most 2.5 times the number of canceled edges in the IO graph: $0 \leq 2.5 \cdot 0$.

- **Case IIIb:** Suppose the next request is to item 2. Two edges are removed from the IO graph. After the IO graph is updated, it contains no edges, meaning that every vertex of the LS graph is consistent with the IO graph. Therefore, the poles only need to move as far apart as possible from each other. There are multiple ways to achieve this while minimizing the total number of edges traversed, but the minimum total movement is 2. For example, the pole at 132 can remain unchanged while the pole at 312 moves to 231, traversing two edges. The expected access cost is $\frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 2 = 2$, since the poles are initially at 132 and 312. The expected paid exchange cost is $\frac{2}{2} = 1$, since the two poles traverse a total of 2 edges, and each pole is followed by BiPolar with probability $\frac{1}{2}$. The potential increases by 2, since the distance between the poles increases from 1 to the maximum possible distance of 3. Therefore, the amortized cost is $2 + 1 + 2 = 5$, which is at most 2.5 times the number of canceled edges in the IO graph: $5 \leq 2.5 \cdot 2$.
- **Case IIIc:** Suppose the next request is to item 3. The analysis for a request to item 3 is symmetric to a request to item 1 (Case IIIa).

- **Case IV:** The IO graph contains three edges (configuration of Figure 6.1IV):
  - **Case IVa:** Suppose the next request is to item 1. The IO graph does not change. Since there is no change to the IO graph, the poles do not need to move. The expected access cost is 0, since both poles are at 123, with item 1 at the front of the ordering. Neither pole incurs a paid exchange, and the potential remains unchanged. Therefore, the amortized cost is

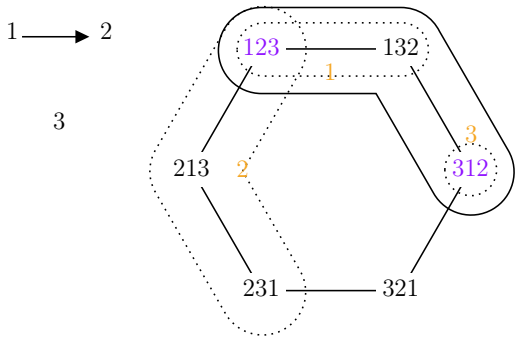
$0 + 0 + 0 = 0$, which is at most 2.5 times the number of canceled edges in the IO graph: $0 \leq 2.5 \cdot 0$.

- **Case IVb:** Suppose the next request is to item 2. The IO graph loses one edge. After the request, one of the poles must move from 123 to 213, while the other pole remains at 123. The expected access cost is 1, since both poles are initially at 123, with item 2 in position 2. The expected paid exchange cost is $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = 0.5$, since one pole moves one edge to reach 213. The potential increases by 1, since the distance between the poles increases from 0 to 1. Therefore, the amortized cost is $1 + 0.5 + 1 = 2.5$, which is at most 2.5 times the number of canceled edges in the IO graph: $2.5 \leq 2.5 \cdot 1$.
- **Case IVc:** Suppose the next request is to item 3. The IO graph loses two edges. After the request, one of the poles must move from 123 to 312, while the other pole remains at 123. The expected access cost is 2, since both poles are initially at 123, with item 3 in position 3. The expected paid exchange cost is $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 2 = 1$, since one of the poles must move two edges to reach 312. The potential increases by 2, since the distance between the poles increases from 0 to 2. Therefore, the amortized cost is $2 + 1 + 2 = 5$, which is at most 2.5 times the number of canceled edges in the IO graph: $5 \leq 2.5 \cdot 2$.

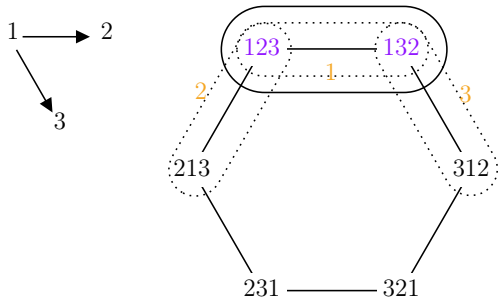
- **Case V:** The IO graph contains no edge (configuration of Figure 6.1V):
  Regardless of the request, no edges will be removed, since the IO graph contains no edges. Before the request, BiPolar's two poles are as far apart as possible. Therefore, they must correspond to reverse orderings. Thus, if one pole places

an item x in position 1, the other places x in position 3. Consequently, the sum of the access costs of the two poles is always 2. Therefore, the expected access cost is $\frac{2}{2} = 1$, since BiPolar follows each pole with probability $\frac{1}{2}$. After a request for item x, as shown in the figure, the poles must move to some neighboring vertices in the LS graph. Moreover, the two opposite configurations can be transformed into any pair of neighboring list states by traversing a total of 2 edges. Hence, the expected paid exchange cost is $\frac{2}{2} = 1$. Once the poles become neighbors, the potential decreases from 3 to 1, resulting in a decrease of 2. Therefore, the amortized cost is $1 + 1 - 2 = 0$, which is at most 2.5 times the number of canceled edges in the IO graph: $0 \leq 2.5 \cdot 0$.

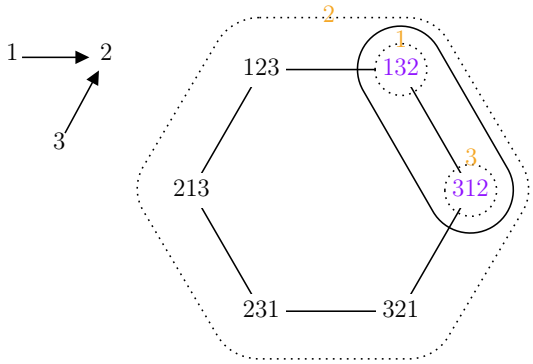
Therefore, in all cases, the amortized cost of BiPolar is within a factor 2.5 of the number of canceled edges in the IO graph. Thus, by Lemma 6.3.1 and Observation 6.3.2, BiPolar is 2.5-competitive for lists of size 3.



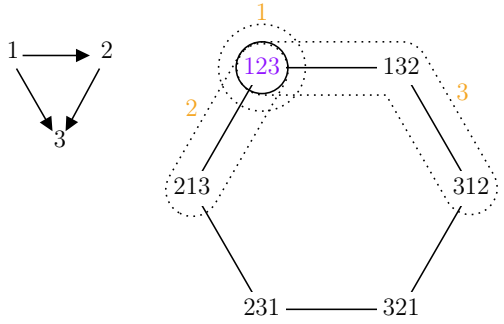
(I) The IO graph contains one edge.



(II) The IO graph contains two edges with the same starting point.



(III) The IO graph contains two edges with the same ending point.



(IV) The IO graph contains three edges

(V) The IO graph contains no edge

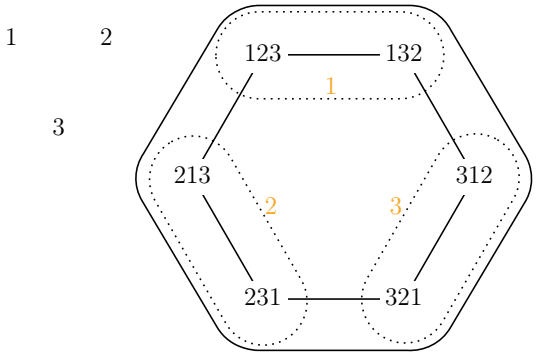


Figure 6.1: An illustration of the case analysis of BiPolar for lists of length 3. Each of the five subfigures indicates one possible IO graph (directed graphs) along with the LS graph and consistent states before a request (solid enclosed areas) and after a request (dotted enclosed areas).

Lists of Size 4

For the lists of length 4, there are 16 distinct partial orderings of the IO graph. To prove an upper bound for the competitive ratio of BiPolar, we use the same potential function as the proof for lists of length 2 and 3; that is, we define the potential to be the distance of the two poles maintained by BiPolar. For each configuration, we used a computer-assisted proof to verify that, for every possible request, the amortized cost, is at most 2.5 times the number of canceled edges in the IO graph. Therefore, we can devise an upper bound of 2.5 for the competitive ratio of BiPolar on lists of size 4. For reference, Figure 6.2 shows the LS graph for lists of size 4, on which the two poles move.

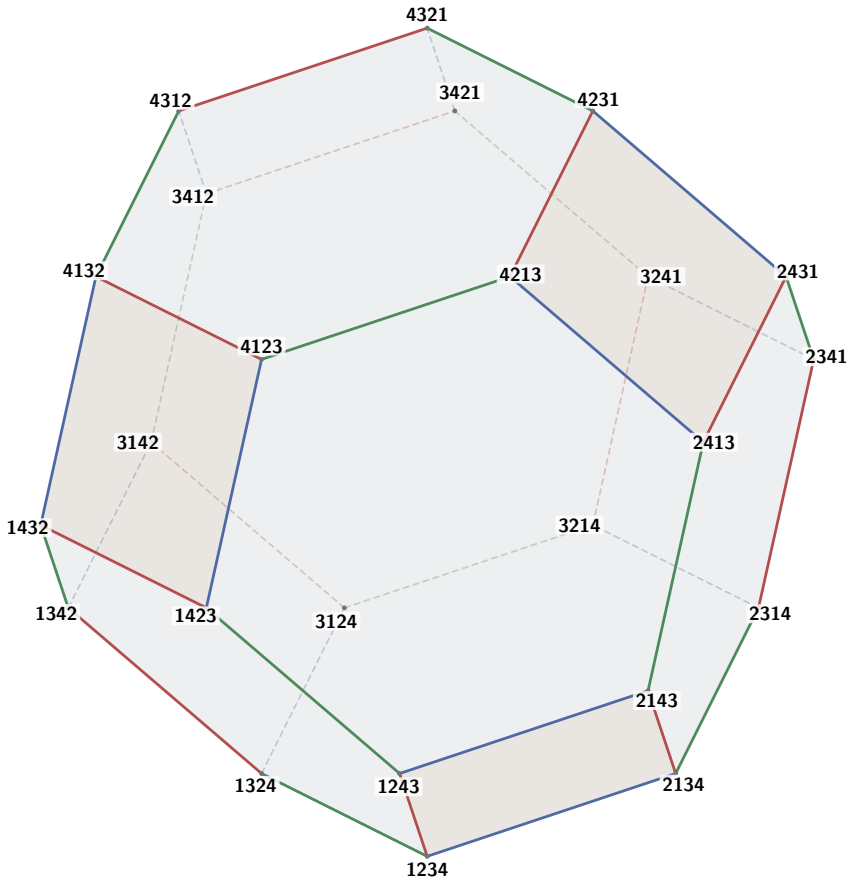


Figure 6.2: The List State (LS) graph for lists of size 4.

6.4 A Remark for Lists of Size $\ell \geq 5$

We used the same method as we did for the lists of size 4 to establish an upper bound for the competitive ratio of BiPolar. Unfortunately, using these techniques, we were not able to establish upper bounds better than the generic algorithm AltComb of Chapter 5. This does not, however, rule out a promising competitive ratio for BiPolar; it could simply be an artifact of the particular analysis and potential function we used.

Chapter 7

Concluding Remarks

In this thesis, we studied the competitive performance of algorithms for the uniform list update problem under both the full and partial cost models. We established an upper bound of $34/13 \approx 2.615$ on the competitive ratio in both cost models, and a lower bound of 2.5 for the partial cost model. For lists of size 2, 3, and 4, we introduced a novel algorithm, BiPolar, and showed that it achieves a competitive ratio of 2.5 for these small list sizes, matching the lower bound for the partial cost model.

Several interesting questions remain open. It would be natural to investigate whether the lower bound of 2.5 established for the partial cost model (Theorem 4.3.2) can be extended to the full cost model. It also remains open whether BiPolar can maintain a competitive ratio close to 2.5 for larger lists under the uniform cost model. The intuition behind BiPolar is to maintain, as closely as possible, the optimal ordering for every pair of items. This intuition can potentially be extended to other cost models, providing a natural direction for studying BiPolar in these settings.

The intuition behind AltComb is to randomly choose between two algorithms, where each algorithm performs well on phases of the input on which the other performs poorly. Through this initial randomization, AltComb achieves good expected performance over any input. A natural extension would be to include additional algorithms in this randomization scheme, which could potentially lead to improved competitive ratios.

Another approach to combining algorithms for Metrical Task Systems, which include the list update problem, was introduced by Bulm and Burch [40]. The idea is to assign higher probability to the candidate algorithm that would have performed best on the input revealed so far and to match its state. For example, to mix Bit and TS, we can maintain both algorithms virtually while having the actual list follow the state of one of them. As more of the input is revealed, if the selected algorithm begins to perform poorly relative to the other, the actual list is likely to switch to following the other algorithm.

In terms of the competitive ratio, this approach for combining Bit and TS would result in exactly the same competitive ratio as AltComb. However, from a probabilistic perspective, the Blum-Burch approach could be expected to perform better than AltComb. If AltComb chooses the wrong algorithm at the beginning, it never reconsiders its choice, whereas the Blum-Burch approach can switch between the two algorithms as more of the input is revealed. However, this flexibility comes at the cost of requiring additional random bits, meaning that the resulting algorithm is no longer barely random.

Overall, the results of this thesis narrow the gap between the best known upper and lower bounds for randomized algorithms in the uniform cost model. These results,

along with the performance of BiPolar, suggest several interesting directions for future work. In particular, it would be useful to determine whether the gap between the upper and lower bounds can be closed, and whether the ideas developed in this thesis can be extended to larger lists and other cost models. Answering these questions could help us better understand the competitive complexity of the list update problem.

Bibliography

[1] Susanne Albers and Maximilian Janke. New bounds for randomized list update in the paid exchange model. In 37th International Symposium on Theoretical Aspects of Computer Science (STACS 2020), pages 12–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[2] Nick Reingold, Jeffery Westbrook, and Daniel D Sleator. Randomized competitive algorithms for the list update problem. Algorithmica, 11(1):15–32, 1994.

[3] John McCabe. On serial files with relocatable records. Operations Research, 13 (4):609–618, 1965.

[4] Shahin Kamali and Alejandro López-Ortiz. A survey of algorithms and models for list update. In Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday, pages 251–266. Springer, 2013.

[5] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. Communications of the ACM, 28(2):202–208, 1985.

[6] Mateusz Basiak, Marcin Bienkowski, Martin Böhm, Marek Chrobak, Łukasz Jeż,

Jiří Sgall, and Agnieszka Tatarczuk. A 3.3904-competitive online algorithm for list update with uniform costs. arXiv preprint arXiv:2503.17264, 2025.

[7] Susanne Albers. Improved randomized on-line algorithms for the list update problem. SIAM Journal on Computing, 27(3):682–693, 1998.

[8] Ran El-Yaniv. There are infinitely many competitive-optimal online list accessing algorithms. Discussion Papers, 6 1996.

[9] Allan Borodin and Ran El-Yaniv. Online computation and competitive analysis. cambridge university press, 2005.

[10] Sandy Irani. Two results on the list update problem. Information Processing Letters, 38(6):301–306, 1991.

[11] Susanne Albers, Bernhard Von Stengel, and Ralph Werchner. A combined bit and timestamp algorithm for the list update problem. Information Processing Letters, 56(3):135–139, 1995.

[12] Boris Teia. A lower bound for randomized list update algorithms. Information Processing Letters, 47(1):5–9, 1993.

[13] Christoph Ambühl, Bernd Gärtner, and Bernhard Von Stengel. A new lower bound for the list update problem in the partial cost model. Theoretical Computer Science, 268(1):3–16, 2001.

[14] Christoph Ambühl, Bernd Gärtner, and Bernhard von Stengel. Optimal lower bounds for projective list update algorithms. ACM Transactions on Algorithms (TALG), 9(4):1–18, 2013.

[15] Susanne Albers, Bernhard von Stengel, and Ralph Werchner. List update posets. Unpublished, available at http://www. maths. lse. ac. uk/Personal/stengel/TEXTE/listupdateposets. pdf, 1996.

[16] Nick Reingold and Jeffery Westbrook. Off-line algorithms for the list update problem. Information Processing Letters, 60(2):75–80, 1996.

[17] Alejandro López-Ortiz, Marc P Renault, and Adi Rosén. Paid exchanges are worth the price. In 32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015), pages 636–648. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015.

[18] Conrado Martınez and Salvador Roura. On the competitiveness of the move-to-front rule. Theoretical Computer Science, 242(1-2):313–325, 2000.

[19] Alexander Golynski and Alejandro López-Ortiz. Optimal strategies for the list update problem under the mrm alternative cost model. Information Processing Letters, 112(6):218–222, 2012.

[20] Mark Manasse, Lyle McGeoch, and Daniel Sleator. Competitive algorithms for on-line problems. In Proceedings of the twentieth annual ACM symposium on Theory of computing, pages 322–333, 1988.

[21] J Ian Munro. On the competitiveness of linear search. In European symposium on algorithms, pages 338–345. Springer, 2000.

[22] Spyros Angelopoulos, Reza Dorrigiv, and Alejandro López-Ortiz. List update with locality of reference. In Latin American Symposium on Theoretical Informatics, pages 399–410. Springer, 2008.

[23] Reza Dorrigiv, Martin R Ehmsen, and Alejandro López-Ortiz. Parameterized analysis of paging and list update algorithms. In International Workshop on Approximation and Online Algorithms, pages 104–115. Springer, 2009.

[24] Jon Louis Bentley, Daniel D Sleator, Robert E Tarjan, and Victor K Wei. A locally adaptive data compression scheme. Communications of the ACM, 29(4): 320–330, 1986.

[25] Reza Dorrigiv, Alejandro López-Ortiz, and J Ian Munro. An application of self-organizing data structures to compression. In International Symposium on Experimental Algorithms, pages 137–148. Springer, 2009.

[26] Shahin Kamali and Alejandro López Ortiz. Better compression through better list update algorithms. In 2014 Data Compression Conference, pages 372–381. IEEE, 2014.

[27] Christoph Ambühl. Offline list update is NP-hard. In European Symposium on Algorithms, pages 42–51. Springer, 2000.

[28] Srikrishnan Divakaran. An optimal offline algorithm for list update. arXiv preprint arXiv:1404.7638, 2014.

[29] Susanne Albers. A competitive analysis of the list update problem with lookahead. Theoretical Computer Science, 197(1-2):95–109, 1998.

[30] Yossi Azar, Shahar Lewkowicz, and Danny Vainstein. List update with delays or time windows. arXiv preprint arXiv:2304.06565, 2023.

[31] Joan Boyar, Shahin Kamali, Kim S Larsen, and Alejandro López-Ortiz. On the list update problem with advice. Information and Computation, 253:411–423, 2017.

[32] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. Journal of the ACM (JACM), 68(4):1–25, 2021.

[33] Yossi Azar, Shahar Lewkowicz, and Varun Suriyanarayana. List update with prediction. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 39, pages 15436–15444, 2025.

[34] Allan Borodin, Nathan Linial, and Michael E Saks. An optimal on-line algorithm for metrical task system. Journal of the ACM (JACM), 39(4):745–763, 1992.

[35] Sébastien Bubeck, Michael B Cohen, James R Lee, and Yin Tat Lee. Metrical task systems on trees via mirror descent and unfair gluing. SIAM Journal on Computing, 50(3):909–923, 2021.

[36] Avrim Blum, Howard Karloff, Yuval Rabani, and Michael Saks. A decomposition theorem for task systems and bounds for randomized server problems. SIAM Journal on Computing, 30(5):1624–1661, 2000.

[37] Sébastien Bubeck, Christian Coester, and Yuval Rabani. The randomized k-server conjecture is false! In Proceedings of the 55th Annual ACM Symposium on Theory of Computing, pages 581–594, 2023.

[38] Maurice G Kendall. A new measure of rank correlation. Biometrika, 30(1-2): 81–93, 1938.

[39] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity. In Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pages 222–227, 1977. doi: 10.1109/SFCS.1977.24.

[40] Avrim Blum and Carl Burch. On-line learning and the metrical task system problem. In Proceedings of the tenth annual conference on Computational learning theory, pages 45–53, 1997.