DISS. ETH No. 14529, 2002

On the List Update Problem

A dissertation submitted to the
Swiss Federal Institute of Technology, ETH Zürich
for the degree of Doctor of Technical Sciences

presented by
Christoph Ambühl
Dipl. Informatik-Ing. ETH
born 15.9.1973, citizen of Willisau-Stadt, Switzerland

accepted on the recommendation of
Prof. Dr. Emo Welzl, ETH Zürich, examiner
Prof. Dr. Susanne Albers, Universität Freiburg i.B., co–examiner
Dr. habil. Bernhard von Stengel, London School of Economics, co–examiner

Meinen Eltern

Abstract

An unsorted linear list is one of the simplest data structures on which one can perform insertions, deletions and lookups. To perform a lookup, the list has to be traversed linearly until the requested item is found. The performance of this data structure can be enhanced by making it self-organizing. In general, the most recently requested item will be moved closer to the front of the list. This is motivated by the empirical observation that, in many cases, requests to items are clustered over time.

An algorithm that updates the list based on the current and past requests is called a list update algorithm. These algorithms are called *online* since they do not know what the forthcoming requests will be.

A very simple algorithm is called `MOVE TO FRONT (MTF)`. Here, the most recently requested item is moved to the front just after the lookup. In 1985, Sleator and Tarjan proved 2-competitiveness of `MTF` which is defined as follows: For any sequence of requests, the running time of `MTF` is at most twice the running time of the optimal *offline* algorithm `OPT`. Compared to online algorithms, offline algorithms are therefore more powerful since they know the whole request sequence in advance.

Using randomized techniques, one can find algorithms which are even more competitive. The best algorithm known to date is the 1.6-competitive `COMB` algorithm due to Albers, von Stengel, and Werchner. It is known that no algorithm can be better than 1.5-competitive.

The ultimate goal is, of course, to find the optimally competitive list update algorithm. All results in this thesis are aimed to give more insight into the structure of the list update problem.

The first result shows that, in the partial cost model, no algorithm can be better than 1.50115-competitive. This is the first non-trivial lower bound in this model. The partial cost model is much easier to analyze. Furthermore, any c-competitive algorithm in the partial cost model is also c-competitive in the standard model.

The second result gives a characterization of all projective algorithms.

They are basically the only kind of algorithms which can be analyzed so far. To prove that a projective algorithm is c-competitive, one only has to prove this on lists with two items which is, of course, much easier. Using this characterization, we give a matching lower bound for projective algorithms in the partial cost model.

The third result shows that it is $\mathcal{NP}$-hard to compute OPT. Hence, there is probably no efficient implementation of OPT. Furthermore, there is only little hope that a better understanding of OPT might give new insights into the list update problem.

Zusammenfassung

Unsortierte lineare Listen gehören zu den einfachsten Datenstrukturen, auf denen sich die Operationen Einfügen, Löschen und Suchen ausführen lassen. Um ein Element in der Liste zu finden muss diese linear durchsucht werden, bis man auf das gesuchte Element stösst. Die Effizienz kann gesteigert werden, indem man die Elemente in der Liste immer wieder umordnet. Man spricht von *selbstorganisierenden Datenstrukturen*. Im Allgemeinen wird das gerade angefragte Element in der Liste weiter nach vorne verschoben, da in vielen praktischen Anwendungen die Anfragen auf die Elemente zeitlich gehäuft sind.

Wir bezeichnen einen Algorithmus, welcher die Liste umordnet, als *list update Algorithmus*. Es handelt sich hier um *online* Algorithmen, da der Algorithmus die zukünftigen Anfragen an die Datenstruktur nicht kennt.

`MOVE TO FRONT (MTF)` gehört zu den einfachsten Algorithmen. Bei jeder Anfrage wird das angefragte Element an den Anfang der Liste verschoben. Sleator und Tarjan zeigten 1985, dass `MTF` 2-kompetitiv ist. Dies bedeutet, dass für jede Sequenz von Operationen die Laufzeit von `MTF` höchstens doppelt so lang ist wie die Laufzeit des optimalen *offline* Algorithmus `OPT`. Im Gegensatz zu online Algorithmen kennen offline Algorithmen die gesamte Sequenz von Beginn weg und sind damit sogar mächtiger als online Algorithmen.

Der beste bekannte Algorithmus ist der 1.6-kompetitive `COMB` (Albers, von Stengel, Werchner). Weiter ist bekannt, dass kein Algorithmus besser als 1.5-kompetitiv sein kann. Das Hauptziel des list update problems besteht darin, den besten Algorithmus zu finden.

Das erste Resultat der Arbeit zeigt, dass kein Algorithmus im $i-1$ Modell besser als 1.50115-kompetitiv sein kann. Dies ist die erste nicht triviale untere Schranke in diesem Modell. Im $i-1$ Modell kostet ein Zugriff auf das $i$te Elemement der Liste nur $i-1$ Zeiteinheiten, während im standard Modell dafür i Einheiten bezahlt werden müssen. Im $i-1$ Modell lassen sich Algorithmen einfacher analysieren und die Analysen lassen sich dann direkt auf das standard Modell übertragen.

Als zweites Resultat werden alle projektiven Algorithmen charakterisiert. Bis auf eine Ausnahme gehören alle bis jetzt analysierten Algorithmen zu dieser Klasse. Um zu zeigen, dass ein projektiver Algorithmus c-kompetitiv ist, reicht es zu zeigen, dass er dies auf Listen mit nur zwei Elementen ist. Dies ist natürlich viel einfacher als im allgemeinen Fall. Mit Hilfe dieser Charakterisierung wird sodann eine untere Schranke von 1.6 für die Kompetitivität von projektiven Algorithmen gezeigt. `COMB` ist also ein optimaler projektiver Algorithmus.

Im letzten Kapitel wird gezeigt, es die Berechnung der optimalen offline Kosten ein $\mathcal{NP}$-hartes Problem darstellt. Dies bedeutet, dass wahrscheinlich kein effizienter Algorithmus dafür exisitert. Es besteht damit wenig Hoffnung, dass ein besseres Verständnis von `OPT` zu effizienteren online Algorithmen führen würde.

Acknowledgments

I wish to express my gratitude to my adviser Prof. Emo Welzl for the opportunity to work in his research group. Thanks for the trust and for the freedom.

Thanks to Susanne Albers and Bernhard von Stengel for agreeing to be my co-advisers. Special thanks to Bernhard von Stengel for inviting me to London, for all the cups of tea we enjoyed in his kitchen, his very careful proof reading and lots of advices.

Whenever I was stuck in my research, talking to Bernd Gärtner was always very enlightening and I left his office with new courage and motivation. His insistence on formality was sometimes annoying, but certainly most valuable.

Many thanks to Michael Hoffmann for providing me with so many flops and his patience in answering my notorious questions.

I'm very grateful to all the people in the "Krauts and K&K" floor for the nice four years I shared with them. Namely (in order of appearance) Tanja Krenn, Jochen Giesen, Johannes Blömer, Lutz Kettner, Martin Will, Michael Hoffmann, Artur Andrzejak, Beat Trachsler, Alexx Below, Falk Tschirschnitz, Udo Adamy, Matthias John, Alexander May, Samuele Pedroni, Uli Wagner, József Solymosi, Sven Schönherr, Susanne Böhm, Csaba Tóth, Floris Tschurr, Tibor Szabo, Franziska Hefti-Widmer, Jochen Giesen, Ingo Schurr, and Bettina Speckmann.

Thanks to Stephan Eidenbenz and Stefan Schnetzler for all the competitions in the squash court and the lifts to Fällanden.

I am also grateful to my parents for their constant support and to Anna Taberska for being my sisterly friend.

Finally, thanks to B., who does not want to appear here by her full name, since technically, she appeared too late.

Contents

Chapter 1

Introduction

1.1 The Sleator Tarjan Result

One of the simplest ways to implement a dictionary is an unsorted linear list. Here, the time needed in order to insert an item into a list containing l items is $l+1$ since the entire list has to be scanned in order to prevent duplicates. The time required for deleting or accessing an item at position i in the list is i units.

Many programmers try to speed up their data structure by reorganizing the items in the list. Usually, a policy is implemented that moves a requested item closer to the front of list in order to save access time on the next request to this item. If one assumes that the items are requested uniformly at random, such ideas can of course not improve the data structure. Still, in most applications, requests will be clustered. For example in the case of a parser for Pascal source code, the keywords of the Pascal programming language will appear very frequently throughout the whole program, while local variables live only in a limited part of the program, but may have an even higher frequency there.

Let us restrict to so-called free exchanges in this section. That is, after item x was accessed, one is allowed to move x to any position closer to the front of the list. This shall be instantaneous, hence we do not charge any time for this update step.

The classical list update algorithms are `Move To Front (MTF)`, `Transpose`, and `Frequency Count`. `MTF` moves the requested item to the front of the list. `Transpose` lets the requested item swap positions with its predecessor. In contrast to the previous algorithms, `Frequency Count` maintains additional information in its items in order to perform the updates. Namely, every item owns a counter which keeps track of the number of requests performed to it so far. Using this information, the items are maintained in non-increasing order of the counters in the list.

In 1985, Sleator and Tarjan [31] gave a theoretical explanation for the empirical finding that `MTF` in general performs best. Theorem 1.1 made competitive analysis and online algorithms a very popular subject in theoretical computer science. The term *competitive analysis* was introduced by Karlin, Manasse, Rudolph, and Sleator in [22].

Theorem 1.1 *No algorithm is faster than `MTF` by more than a factor of* $(2 - \frac{1}{n})$ *on any sequence of insertions, deletions, and lookups, where* n *is the maximal number of items ever contained in the list.*

Proof. In order to prove the theorem, let `A` be the algorithm `MTF` competes against.

Let σ be a sequence of m requests to be performed in turn. The requests are either lookups, deletions or insertions. Let further L_i^A be the list state the algorithm `A` maintains just before it serves the ith request of σ. The states L_i^{MTF} are defined analogously. Both algorithms start from the same initial list state. That is, we have $L_0^{\mathsf{A}} = L_0^{\mathsf{MTF}}$.

The proof is based on a potential function Φ that translates the joint list states of `MTF` and `A` to a natural number. Let t_i be the time `MTF` spends on the ith request and t_i^{A} the time of `A` for the same request. We define the amortized cost

$$a_i := t_i + \Delta\Phi_i = t_i + \Phi_i - \Phi_{i-1}. \tag{1.1}$$

If we can show that

$$a_i \leq (2 - \frac{1}{n}) \cdot t_i^{\mathsf{A}}, \tag{1.2}$$

we are done because the total runtime of MTF can be written as

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i - \Phi_m + \Phi_0.$$

With $\Phi_0 = 0$ and $\Phi_m \geq 0$, we conclude

$$\sum_{i=1}^{m} t_i \leq (2 - \frac{1}{n}) \cdot \sum_{i=1}^{m} t_i^{\mathsf{A}}.$$

The potential function Φ that allows us to prove (1.2) is based on the relation between the list states of the two algorithms. More precisely, Φ_i is the number of inversions between the two list states L_i^{A} and L_i^{MTF}. An inversion is a pair of items whose relative order differs in both lists.

In order to prove (1.2), we have to distinguish between lookups, insertions and deletions. Each of the three cases has a successful and an unsuccessful subcase. The most interesting case is when the ith request is a successful lookup to an item x. Figure 1.1 shows the two list states

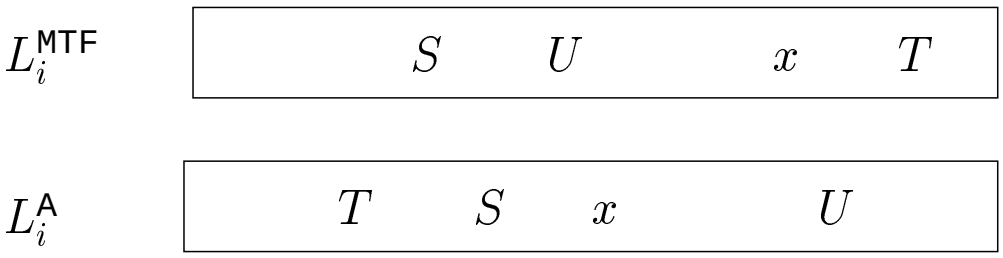

Figure 1.1: *List state before the ith request*

just before the lookup. By S we denote the set of items which are in front of x in both lists. T denotes the items which are behind x in L_i^{MTF} but in front of x in L_i^{A}. Finally, U contains the set of items which are in front of x in L_i^{MTF} and behind x in L_i^{A}. We have

$$\begin{aligned} t_i^{\mathsf{MTF}} &= |S| + |U| + 1, \\ t_i^{\mathsf{A}} &= |S| + |T| + 1, \\ \Delta\Phi_i &\leq |S| - |U|, \end{aligned}$$

the latter because the change in potential only affects inversions involving x. The inequality holds with equality if x only moves in L_i^{MTF} of Figure 1.1. If A also moves x closer to the front, $\Delta\Phi_i$ becomes even smaller. Plugging this into 1.1, we obtain

$$\begin{aligned} a_i &= t_i + \Delta\Phi_i \leq |S| + |U| + 1 + |S| - |U| \\ &= 2 \cdot |S| + 1 \leq 2 \cdot t_i^{\mathsf{A}} - 1 \leq (2 - \frac{1}{n}) \cdot t_i^{\mathsf{A}}. \end{aligned}$$

The case of an unsuccessful insertion is similar to a lookup. An unsuccessful deletion or lookup costs the same for both MTF and A while the potential function remains unchanged. Thus in this case, we have $a_i \leq t_i^{\mathsf{A}}$. In the case of a successful insertion (items are inserted at the end of the list), we again have $t_i = t_i^{\mathsf{A}}$. Since then $\Delta\Phi_i \leq t_i - 1$, we obtain $a_i = t_i + \Delta\Phi_i \leq 2 \cdot t_i^{\mathsf{A}} - 1$. □

1.2 The List Update Model

The subject of the list update problem is to find algorithms which beat the constant $2 - \frac{1}{n}$ of MTF. In order to do that, we have to define the model more rigorously.

In the remainder of this thesis, we will stick to the static list update problem. This means that we start from an initial list state containing n items. Instead of insertions, deletions, and lookups, the only considered requests are lookups to one of the n items in the list. In general, we will use the term "request to x" to denote a lookup of item x.

List states will be denoted in brackets with the items ordered from left to right. Usually, L stands for list states but also for the set of items in the list. Request sequences are denoted by σ and their length by m.

The algorithms we consider are online algorithms. That is, their behavior can depend only on the current and the past requests, but not on future requests. MTF clearly is online. Since we would like to implement our algorithms to speed up our dictionary, the online property is crucial because in general the algorithm has no knowledge about future requests.

On the other hand, the proof of Theorem 1.1 does not require the algorithm A to be online. MTF has to behave well on any request sequence and against any algorithm. Given σ, we can design an online algorithm A such that it behaves optimally on σ. Therefore we can assume that A is an optimal offline algorithm.

We will measure the runtime of the algorithms by cost units we charge to the different operations.

Definition 1.2 *Let $A(\sigma)$ be the cost an online algorithm A spends to process a request sequence σ and let $OPT(\sigma)$ be the cost the optimal offline algorithm spends for that task. We say A is c-competitive if there exists a constant b such that for all request sequences σ*

$$A(\sigma) \leq c \cdot OPT(\sigma) + b \tag{1.3}$$

holds. In general, we will treat the number of items n as constant. Hence, b is allowed to depend on n. An algorithm is called competitive if it is c-competitive for some real number c.

A is strictly c-competitive if for all request sequences σ we have

$$A(\sigma) \leq c \cdot OPT(\sigma). \tag{1.4}$$

For the case of randomized algorithms, the term $\mathsf{A}(\sigma)$ denotes the expected cost A spends on the sequence σ.

The cost model that we will stick to for the rest of this thesis is the *partial cost model*, meaning that we only pay $i - 1$ units in order to access the item at position i in the list. It turns out that this model is much easier to analyze than the *full cost model* we used in Section 1.1. Still, if an algorithm A is c-competitive in the partial cost model, it is also c-competitive in the full cost model. Usually with some dependence on n, the number of items in the list. To see this, note that because of $c \geq 1$, inequalities (1.3) and (1.4) remain valid if we subtract $|\sigma|$ from $A(\sigma)$ and $\mathsf{OPT}(\sigma)$ on both sides.

In the partial cost model, MTF is 2-competitive. Indeed, MTF is an optimal deterministic algorithm. Algorithms which beat the competitive ratio of 2 for arbitrary long lists need to be randomized.

Concerning the updates, we distinguish between free exchanges and paid exchanges. At the expense of one cost unit, any consecutive pair of items can swap its order at any time. On the other hand, just after a request to an item x, this item may be moved to a position closer to the front of the list without cost.

Note that free exchanges can be modeled by paid exchanges. Instead of first paying k units in order to access item x and then move it at no charge t positions closer to the front, one can first move the item t positions and then access the item. In both cases, we pay exactly k units. Although stated as Theorem 3 in [31], the converse is not true. For an example, let $L = [abc]$ and $\sigma = cbbc$. Here, an optimal algorithm moves a behind b and c before the first request to c. This requires paid exchanges. This can be proved formally by projectivity arguments we will encounter later in this section.

Although paid exchanges are more general, most algorithms known to date can be stated such that only free exchanges are used. It is not clear whether paid exchanges can lead to better algorithms. On the other hand, proofs and definitions often become more elegant if only paid exchanges are considered.

If we forget about free exchanges, we can specify any deterministic online algorithm A by a function

$$S^{\mathsf{A}} : \Sigma \to \mathcal{L}.$$

Here Σ denotes the set of request sequences, whereas $\mathcal{L}$ denotes the set of the $n!$ states the n items can attain. $S^{\mathsf{A}}(\sigma)$ denotes the list state in which the last request of σ is performed if algorithm A is used. Using this notation, the initial list state can be denoted by $S^{\mathsf{A}}(\emptyset)$. We will omit the A in $S^{\mathsf{A}}(\sigma)$ when the algorithm used is determined by the context.

The list update problem can be stated in terms of game theoretic concepts, namely as an infinite two-person zero-sum game. The first player is called *adversary*. His pure strategies are the set of finite request sequences. The strategies of the second player, called online player, are the set of deterministic online algorithms. Let b be a fixed constant and let (σ, A) be a pair of pure strategies for the adversary and the online

player. Then the payoff is

$$\frac{\mathsf{A}(\sigma)}{\mathsf{OPT}(\sigma) + b}.$$

The goal of the online player is to choose A such that

$$\sup_{\sigma} \frac{\mathsf{A}(\sigma)}{\mathsf{OPT}(\sigma) + b} \tag{1.5}$$

is minimized. On the other hand, the adversary chooses σ such that

$$\inf_{\mathsf{A}} \frac{\mathsf{A}(\sigma)}{\mathsf{OPT}(\sigma) + b} \tag{1.6}$$

is maximized. If the set of strategies of the two players was finite, an application of the famous minimax theorem would prove the existence of a pair of randomized strategies for which (1.5) and (1.6) are equal. This is called an equilibrium and the corresponding value is referred to as *the value of the game*. Although the minimax theorem cannot be applied in our case, we prove in Section 2.8 that the list update problem indeed has a value. However, approximating it via brute force computation is hopeless.

A c-competitive algorithm proves that the value of the game cannot be larger than c, while randomized strategies for the adversary can give lower bounds on the game value. For the full cost model, the known lower and upper bounds are 1.5 and 1.6. We will give an improved lower bound of 1.50115 for the partial cost model in Chapter 2 of this thesis.

A randomized strategy is defined by a probability distribution over the set of all deterministic strategies. Hence in the case of the online player, randomized algorithms can be defined as follows. Before serving the first request, one of the deterministic strategies is chosen according to the probability distribution. This strategy is then used for the whole request sequence. A randomized strategy of the adversary is just a probability distribution over the set Σ.

This game theoretic model nicely covers the notion of an oblivious adversary. The adversary cannot observe the random choices made by the

online algorithm. If he could do so, his pure strategies would depend on the current list state of the online algorithm. This kind of adversaries are called adaptive. It is easy to see that in this case, randomization is pointless and no algorithm can be better than 2-competitive. Hence, MTF is optimally competitive against adaptive adversaries.

1.3 Projective Algorithms

In order to describe projective algorithms, we have to introduce the concept of projections of request sequences and list states.

Let a request sequence σ be given and fix a pair of items x, y, the projection of σ to x and y is the request sequence σ where all requests which are not to x or y are removed. We denote the projection of σ to x and y by σ_{xy}. Given a list state L, the projection to x and y is obtained by removing all items but x and y from the list. This is denoted by L_{xy}.

Definition 1.3 *Let $S_{xy}(\sigma)$ be the projection of $S(\sigma)$ to x and y. A deterministic algorithm A is projective if for all pairs of items x, y and all request sequences σ we have*

$$S_{xy}(\sigma) = S_{xy}(\sigma_{xy}). \tag{1.7}$$

A randomized algorithm is projective if all deterministic algorithms chosen with positive probability are projective.

In words, an algorithm is projective if the relative position of any pair of items depends only on the initial list state and the requests to x and y in the request sequence.

Already in [13], Bentley and McGeoch observed that MTF has this property. To see that MTF is projective, observe that x is in front of y if and only if y has not been requested yet or if the last request to x took place after the last request to y.

With the exception of Irani's Split algorithm [20, 21], projective algorithms are the only family of algorithms one can analyze so far. The next theorem is responsible for this fact.

Theorem 1.4 *Let A be a (strictly) projective algorithm. If it is c-competitive on lists with two items, it is also (strictly) c-competitive on lists of arbitrary length.*

Proof. Let $\mathsf{A}_{xy}(\sigma_{xy})$ denote the cost the projective algorithm A spends in order to serve σ_{xy} from the initial list $S_{xy}(\emptyset)$, which is the list with only the items x and y, initially ordered like in $S(\emptyset)$.

It holds that

$$\mathsf{A}(\sigma) = \sum_{\{x,y\}\subseteq L} \mathsf{A}_{xy}(\sigma_{xy}). \tag{1.8}$$

To see this, consider the ith request to some item x in σ. Let σ' be the prefix of σ up to this request. And let B be the set of items which are in front of x in $S(\sigma')$. Because of (1.7), we have $S_{xy}(\sigma') = [yx]$ if and only if $y \in B$. Therefore the access cost for any request is the same on both sides.

Concerning update costs, let $\sigma' := \sigma z$ for some request sequence σ and some item $z \in L$. We again use (1.7) to note that $S_{xy}(\sigma) \neq S_{xy}(\sigma')$ if and only if $S_{xy}(\sigma_{xy}) \neq S_{xy}(\sigma'_{xy})$. Hence there is a bijection between the transpositions on both sides. Therefore the update cost is again the same on both sides. A similar idea works for OPT. By $\mathsf{OPT}_{xy}(\sigma_{xy})$ we denote the minimal cost OPT would pay on the sequence σ_{xy} starting from $S_{xy}(\emptyset)$. One (not necessarily optimal) way to serve σ_{xy} is to force for all pairs x, y and all prefixes σ' of σ

$$S^{\mathsf{OPT}_{xy}}(\sigma'_{xy}) := S^{\mathsf{OPT}}_{xy}(\sigma').$$

In this way, (1.8) would also hold for OPT. Hence if we really serve the pair lists optimally,

$$\mathsf{OPT}(\sigma) \geq \overline{\mathsf{OPT}}(\sigma) := \sum_{\{x,y\}\subseteq L} \mathsf{OPT}_{xy}(\sigma_{xy}). \tag{1.9}$$

Since A is c-competitive on two items, we find for every pair of items x, y a constant b_{xy} such that for all σ we have

$$\mathsf{A}_{xy}(\sigma_{xy}) \leq c \cdot \mathsf{OPT}_{xy}(\sigma_{xy}) + b_{xy}.$$

Using this fact we get

$$\begin{aligned} \mathsf{A}(\sigma) &= \sum_{\{x,y\}\subseteq L} \mathsf{A}_{xy}(\sigma_{xy}) \\ &\leq \sum_{\{x,y\}\subseteq L} (c \cdot \mathsf{OPT}_{xy}(\sigma_{xy}) + b_{xy}) \\ &\leq c \cdot \overline{\mathsf{OPT}}(\sigma) + \sum_{\{x,y\}\subseteq L} b_{xy} \\ &= c \cdot \overline{\mathsf{OPT}}(\sigma) + b \qquad (1.10) \\ &\leq c \cdot \mathsf{OPT}(\sigma) + b. \end{aligned}$$

For the strict case, just set all $b_{xy} := 0$. □

Hence the case of pair lists seems to be crucial for the analysis of projective algorithms. Luckily enough, there is a very simple implementation of OPT for this case. See [28] for a proof that the following algorithm is indeed optimal.

Algorithm 1.5 (OPT on two items) *Assume w.l.o.g. that the current list state is $[xy]$. Move y to the front if and only if the upcoming two requests are to y.*

Note that the above algorithm only examines the current request and the next request to determine the optimal move. We say this algorithm uses lookahead one. Already for lists with three items, all future requests may be needed to serve the sequence optimally [28, 2].

Using the fact that MTF is projective, the proof of Theorem 1.1, reduced to the static problem, now becomes very simple.

Proof of Theorem 1.1 (projective version). All we have to show is that MTF is strictly competitive on lists with two items x and y.

Let σ be a request sequence on two items x and y. We break σ into subsequences in an iterative way. We let subsequences end just before OPT has to pay for a request. Hence in every subsequence except the first, OPT has to pay for the first request. All other requests are free

for OPT. The first subsequence is either a regular subsequence or it is free for both algorithms and therefore can be skipped. Regular subsequences are either of the form x^l, y^l, xy^l, or yx^l, $l \geq 0$. On all of these subsequences, MTF pays at most two units. If we add up the cost for both algorithms, we find that

$$\mathsf{MTF}(\sigma) \leq 2 \cdot \mathsf{OPT}(\sigma).$$

□

In Chapter 3, we will give a complete characterization of the set of projective algorithms. A simple subset of these which covers all reasonable projective algorithms is the set of critical request algorithms [11]. As we have already seen, algorithms are defined by functions that translate a request sequence into a list state. In the case of a critical request algorithm, $S(\sigma)$ is obtained as follows.

Algorithm 1.6 (critical request algorithms) *Let us first see how* deterministic *critical request algorithms are defined.*

Every item x in the list has a so-called critical request function

$$F_x : \mathbb{N} \to \mathbb{N}_0, \text{ with } F_x(i) \leq i,$$

where $\mathbb{N} = \{1, 2, 3, \ldots\}$ and $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$. Let $|\sigma_x|$ denote the number of request to x contained in σ. We call the $F_x(|\sigma_x|)$th request to x in σ the critical request to x. Since $F_x(|\sigma_x|)$ can be zero, some items may have no critical request. In $S(\sigma)$, all items with critical request are grouped together in front of the items without critical request. The items with critical requests are ordered according to the time of the $F_x(|\sigma_x|)$th in σ. The later a critical request took place in the sequence, the closer the item is to the front. The remaining items are placed behind, according to their order in the initial sequence.

Randomized critical request algorithms are just a probability distribution over the set of deterministic critical request algorithms.

As an example, let the online algorithm for three items a, b, and c be defined by functions F_a, F_b, and F_c. The table below lists the values for the arguments 1 to 4.

	1	2	3	4
F_a	1	0	3	2
F_b	0	2	3	4
F_c	1	2	2	2

Let the initial list state be $[abc]$. Let us determine $S(\sigma)$ for $\sigma = abbcab$. We have $F_a(|\sigma_a|) = F_a(2) = 0$, hence a does not have a critical request. For b we have $F_b(|\sigma_b|) = 3$, therefore the third request to b in σ is its critical request. For c we have $F_c(|\sigma_c|) = 1$. Thus we have $S(\sigma) = [bca]$ since the third request to b happened after the first request to c. Item a, not having a critical request, must be at the very end.

Algorithms based on critical request functions clearly are projective, since the relative order of any pair of items just depends on the relative order of the requests to x and y in σ and the relative order of x and y in the initial list state.

The currently best list update algorithm is COMB due to Albers, von Stengel, and Werchner [5]. COMB is a combination of two simpler algorithms.

Algorithm 1.7 (COMB) *Before the first request, toss a biased coin to decide which algorithm to use for the whole sequence. Use BIT with probability 0.8, with probability* 0.2 *use TS.*

BIT is an elegant 1.75-competitive algorithm due to Reingold, Westbrook, and Sleator [29]. It is a member of a more general class of algorithms called RANDOM RESET algorithms. The best algorithm in this class is $\sqrt{3}$-competitive.

Algorithm 1.8 (BIT) *Every item maintains a bit. Initially, each bit is set to 0 or 1 using a fair coin. On a request to item* x*, the bit is flipped. Only if the bit changes to* 1*, we move the item to the front. Otherwise the position of* x *is unchanged.*

TS is a deterministic member of the class TIMESTAMP(p) due to Albers [1]. TS is 2-competitive. All TIMESTAMP algorithms are projective. While all previous algorithms either moved the requested item

to the front of the list or left its position unchanged, TIMESTAMP algorithms sometimes move the requested item to a position within the list.

Algorithm 1.9 (TS) *After each request, the accessed item x is moved in front of exactly the items that have been requested at most once since the last request to x. On its first request, every item remains at its position.*

While BIT is well defined, it is not clear whether TS actually defines an algorithm. We first would have to prove that the items that have to be passed by x are situated in a consecutive block just in front of x. Only if this holds, the algorithm can run as described.

Additionally, we would have to prove that both algorithms are projective. At least in the case of TS, this is not trivial. Using the critical requests, both algorithms can be described very easily and in such a way that their projective behavior becomes obvious.

In order to overcome special cases in the description, we first define the concept of an *augmented* request sequence. Given an initial list state $[x_1 x_2 \dots x_n]$ and a request sequence σ, the augmented request sequence is $x_n x_n x_{n-1} x_{n-1} \dots x_1 x_1 \sigma$. The two additional requests for every item will have request number -1 and 0 and will allow critical request functions to attain the values 0 and -1. Note that this prefix will never actually be served. The augmented request sequence is just a concept in order to describe algorithms in a compact way.

With this trick, TS can be described very easily in terms of critical request functions by

$$F_x(i) := i - 1$$

for all items x. Hence the critical request of every item is just the second-to-last request in the augmented request sequence.

As an example, the list state after $\sigma = abbcbba$ with initial list state $[abc]$ is $[bac]$ because of the ordering of the second-to-last requests in $ccbbaaabbcbba$. If we add another request to c, the new list state will be $[bca]$ because the second-to-last request to c is now more recent than the second-to-last request to a.

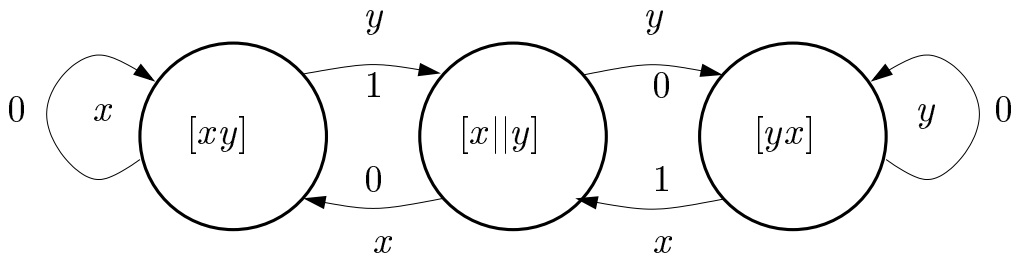

Figure 1.2: *Automaton describing the two items case.*

Since BIT is randomized, the critical requests are also randomized. For every item x, its critical request function can be written as

$$F_x(i) := i - \big((i + b_x) \mod 2\big)$$

where b_x is x's bit initialized by a fair coin. Hence the critical request is the last or the second-to-last request with equal probability. The correspondence to the definition using the bits is the following. The critical request is on the last request if and only if the current value of the bit is 1.

Viewing BIT and TS as critical request algorithms, we can give a simple proof for COMB's competitiveness based on a potential function.

COMB's critical request functions are static in the sense that the probability for the last request in the augmented request sequence to be the critical one is always 0.4, while the second-to-last request has probability 0.6. Hence in order to deduce the expected access cost of a request, all we need is the relative order of the last two requests to each item. Since both BIT and TS use only free exchanges, the update costs are zero.

We want to describe OPT for two items in an online fashion, hence get rid of the lookahead needed in algorithm 1.5. This can be done by a state diagram where we encode into the states the fact that the current list state might depend on a future request.

The diagram in Figure 1.2 indeed does this job. If we are in the left state, the optimal list state is $[xy]$. In the right state, it is $[yx]$. The middle state is the state we move into if the item at the second position

in the current list has been requested. By $[x||y]$, we denote the fact that the items are in some sense parallel since we do not know whether the optimal list state is $[xy]$ or $[yx]$. If the next request is x, the optimal choice was $[xy]$, otherwise $[yx]$. In any case, a request leaving the middle state is free of charge. In order to compute the optimal cost of a request sequence, one just moves in the diagram according to the request sequence like in a deterministic finite automaton. The starting state is either the right or the left state, depending on the initial list. Then $\texttt{OPT}(\sigma)$ is equal to the number of times one moves into the middle state.

With this preparatory work, we are now ready to prove

Theorem 1.10 *COMB is strictly* 1.6*-competitive*

Proof. It turns out that the joint behavior of COMB and OPT can be described by a four state diagram like in Figure 1.3. Each of the four states in labeled by one of the labels of Figure 1.2 and a permutation of $xxyy$, which represents the ordering of the latest two requests to the two items.

These labels allow to compute both COMB's and OPT's cost for the next request. In the case of OPT, we already know from Figure 1.2 how to determine the cost of a new request depending on the state. On the other hand, since COMB has its critical requests either on the last or the second-to-last request, we only need to know the ordering of the two latest requests per item in order to determine the cost of a new request.

Since x and y use the same critical request functions, we can merge states like $\{xxyy, [xy]\}$ and $\{yyxx, [yx]\}$. The starting state is S_0. Any request sequence translates to a path in the automaton starting in S_0. Each request corresponds to a transition. Using the information stored in the states, we can give COMB's as well as OPT's cost for any transition leaving the state. The cost of COMB and OPT is indicated by the small numbers as the pairs COMB/OPT. Note that COMB's cost are expected cost. The cost for serving a request sequence σ for both algorithms is the sum of the cost assigned to the transitions used.

In order to show that COMB is 1.6-competitive, we use a potential func-

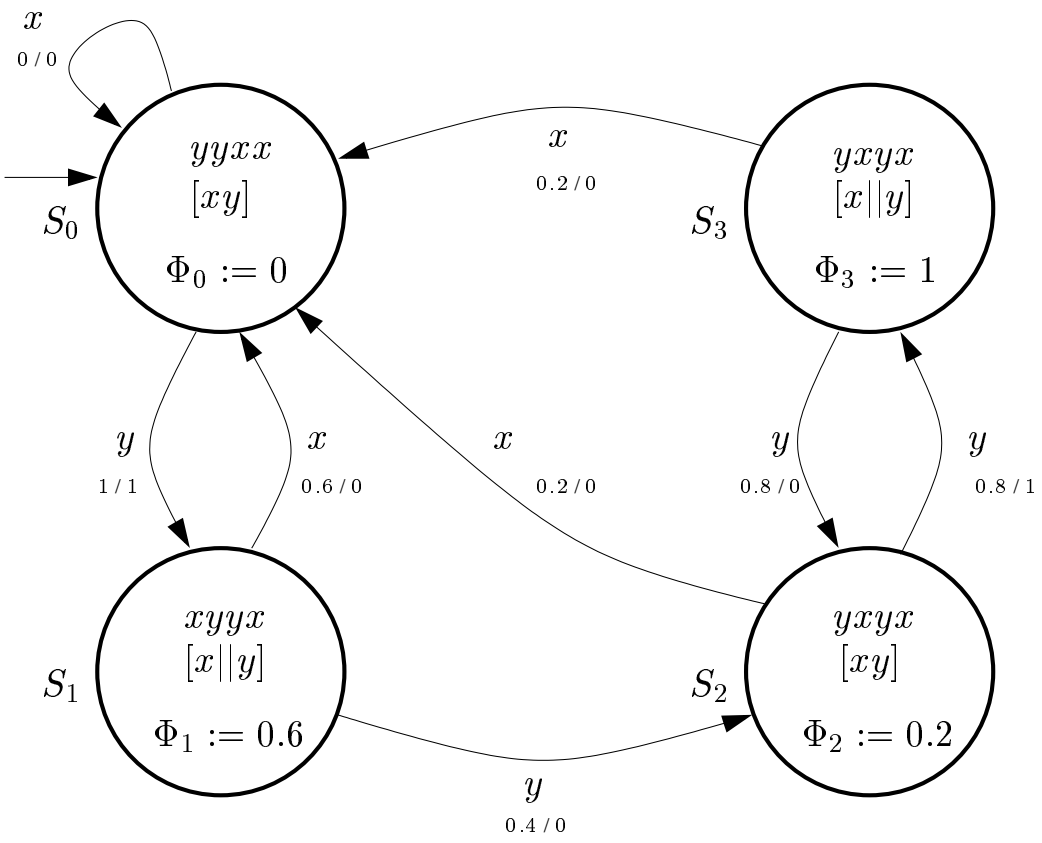

Figure 1.3: *Automaton describing* `COMB` *and* `OPT` *in the partial cost model.*

tion. It is indicated by the values Φ_i for each state S_i. Note that the value Φ_i equals the maximum cost COMB has to pay on a path starting at S_i which is free for OPT. Let t_i and t_i^{OPT} be COMB's and OPT's cost for the ith request in a request sequence σ of length m. Let the ith request move from state S_g to S_h.

The amortized cost is then

$$a_i = t_i + \Delta\Phi = t_i + \Phi_h - \Phi_g.$$

By checking all eight transitions in Figure 1.3, one can prove

$$a_i \leq 1.6 \cdot t_i^{\mathsf{OPT}}. \tag{1.11}$$

Indeed, (1.11) holds with equality except for the transition from S_3 to S_0. Let S_l be the state where σ ends. Since the potential of the starting state S_0 is zero and $\Phi_l \geq 0$, we obtain

$$\begin{aligned}
\mathsf{COMB}(\sigma) &= \sum_{k=1}^{m} t_k \\
&= \sum_{k=1}^{m} a_k - \Delta\Phi_k \\
&= \sum_{k=1}^{m} 1.6 \cdot t_i^{\mathsf{OPT}} - \Phi_l + \Phi_0 \\
&\leq 1.6 \cdot \sum_{k=1}^{m} t_i^{\mathsf{OPT}} \\
&= 1.6 \cdot \mathsf{OPT}(\sigma).
\end{aligned}$$

□

COMB is by far not the only 1.6-competitive algorithm, but it seems to be one of the simplest. Other 1.6-competitive algorithms are obtained by choosing TS, BIT and MTF with different probabilities than COMB does. A different way is to choose randomly for each item whether it

should use the critical request functions of TS, BIT, or MTF. In Chapter 3, we will show that no projective algorithm can beat COMB.

The original proof of Theorem 1.10 is based on a partitioning of the request sequences. The phases are those described in the following lemma.

Lemma 1.11 *Consider a list with the only items x and y with initial state $[xy]$. The following table shows the expected cost for the algorithms BIT, TS, COMB and OPT for a set of request sequences. We assume $l \geq 0$ and $k \geq 1$.*

request sequence	*BIT*	*TS*	*COMB*	*OPT*
$x^l yy$	$\frac{3}{2}$	2	1.6	1
$x^l (yx)^k yy$	$\frac{3}{2}k + 1$	$2k$	$1.6 + 0.8k$	$k + 1$
$x^l (yx)^k x$	$\frac{3}{2}k + \frac{1}{4}$	$2k - 1$	$1.6k$	k

Proof. Note that BIT and TS only use free exchanges. Therefore, we only need to count expected access costs. The expected cost spent for the request to y in the sequence $\sigma y \sigma'$ is the probability that x is in front of y in the list after serving σ. Since the order of the items in the list is determined by the order of the critical requests in the augmented request sequence, this is not a hard task. The following tables give the cost of the sequences. Note that in the tables the augmented request sequences are shown. Thus the actual request sequences start at the fifth request.

	y	y	x	x	x^l	y	y
BIT					0	1	0.5
TS					0	1	1

	y	y	x	x	x^l	y	x	$(y$	$x)^{k-1}$	x
BIT					0	1	0.5	0.75	0.75	0.25
TS					0	1	0	1	1	0

	y	y	x	x	x^l	y	x	$(y$	$x)^{k-1}$	y	y
BIT					0	1	0.5	0.75	0.75	0.75	0.25
TS					0	1	0	1	1	1	0

□

Using Lemma 1.11, the prove of Theorem 1.10 goes as follows.

Proof. We partition every request sequence into subsequences, each of them terminated by two consecutive requests to the same item. Assuming initial list state $[xy]$, the first sequence σ' is one of those described in the lemma. If that subsequence terminates in xx, the next subsequence σ'' will again be of one of the three forms. Note that the cost of σ'' will again be like stated in the lemma because the initial list state for σ'' is again $[xy]$ for BIT, TS and OPT and the double request to x works like an augmentation prefix. If σ' terminates in yy, σ'' will be one of the three forms with x and y interchanged. Again the cost here is the same because also in the initial state and in the 'augmentation prefix', x and y change roles.

According to the table of Lemma 1.11, COMB's cost is bounded by 1.6 times OPT's cost for all these subsequences. But the very last subsequence might not belong to one of the three types. However it will be a prefix of one of the three types with only the last request missing. Since OPT never pays on the last request, adding it would leave the cost of OPTunchanged and merely overestimate the cost of COMB, so the cost ratio of COMB versus OPT is even better in this case.

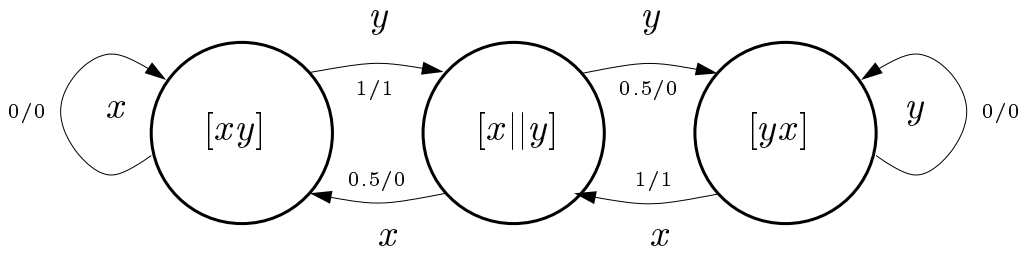

Figure 1.4: *The optimal algorithm on two items.*

Since we have $\mathsf{COMB}(\sigma) \leq 1.6 \cdot \mathsf{OPT}(\sigma)$ for all subsequences, COMB is strictly 1.6-competitive on two items. From Theorem 1.4 it therefore follows that COMB is strictly 1.6-competitive on lists of arbitrary length. □

On two items, there exists a 1.5-competitive online algorithm. Unfortunately, one cannot express it in terms of projective algorithms.

Algorithm 1.12 *The algorithm uses of Figure 1.2 by keeping track of OPT's state in the figure. If OPT is in the left or the right state, both algorithms have the same list state. Whenever OPT moves into the middle state, that is, the item at the second position in the list was requested, the online algorithm moves the item to the front only with probability 0.5.*

The cost of the online algorithm and OPT are given in Figure 1.4. From Figure 1.4 it is easy to see that between two visits of the middle state, OPT pays exactly one unit, whereas our online algorithm pays 1.5 units. Therefore our algorithm is 1.5-competitive. A proof using a potential function would assign potential 0.5 to the state $[x||y]$ and 0 to the others.

On two items, a lower bound of 1.5 is very easily obtained. Let $S(\emptyset) := [xy]$ be the initial list state. If the adversary chooses the request sequences yyy and $yxxx$ with equal probability, no algorithm can be strictly c-competitive with $c < 1.5$. Note that OPT pays one unit on either sequence. To do this, it has to move y to the front in the first sequence, but leave it at the second position in the second sequence.

Otherwise, it pays at least two units. Any online algorithm makes a mistake with probability 0.5. Therefore, its expected cost are 1.5.

For the non-strict case, one has to repeat this process arbitrarily many times. This can be done because one can assume that the list states are equal after one round. Depending on whether it is $[xy]$ or $[yx]$, the next round uses the same sequences again or it uses xxx and $xyyy$.

As we have seen earlier, a c-competitive algorithm in the partial cost model is also c-competitive in the full cost model. Concerning lower bounds, it is the other way round. Lower bounds in the full cost model generalize to the partial cost model.

1.4 The Offline Problem

Since the performance of an online algorithm is compared with the optimal offline algorithm OPT, understanding the problem of computing $\mathsf{OPT}(\sigma)$ becomes an issue itself.

A simple algorithm has running time $O((n!)^2 m)$ on a list with n items and a sequence with m requests. It is based on a straightforward dynamic programming algorithm for metrical task systems [19] which works as follows.

Let $d(i, L)$ be the minimal cost needed to serve the first i requests of the request sequence σ and end up in the list state L. By $\mathcal{L}$ we denote the set of all $n!$ possible list states. Using dynamic programming, we have to fill a table with $m + 1$ rows and $n!$ columns with the values $d(i, L)$, $i = 0 \ldots m$, $L \in \mathcal{L}$. Once the table is filled, $\mathsf{OPT}(\sigma)$ is obtained by

$$\min_{L \in \mathcal{L}} d(m, L).$$

To fill the table, we use the recursion

$$d(i, L) = \min_{L' \in \mathcal{L}} \big(d(i-1, L') + trans(L', L) \big) + acc(i, L). \qquad (1.12)$$

Here, $trans(L', L)$ denotes the minimal cost to move from state L' to L and $acc(i, L)$ denotes the cost for accessing σ_i in L. The base case is

$d(0, L) = trans(L, S(\emptyset))$. The time needed to compute all $d(i, L)$ is $O((n!)^2 m)$.

This runtime can be reduced to $O(2^n n! m)$ by using the fact that there is an optimal algorithm which uses only so-called *subset transfers* [28]. In a subset transfer, one moves a subset of the items preceding the requested item x just behind x without changing their relative orders. Only $O(2^n)$ among the $n!$ possible transformations are subset transfers.

In his semester thesis, Pietrzak [26] showed that the problem can be solved in time $O(n! n^3 m)$. As a first step, we break equation 1.12 into

$$d'(i, L) := \min_{L' \in \mathcal{L}} \big(d(i-1, L') + trans(L', L)\big) \qquad (1.13)$$

$$d(i, L) := d'(i, L) + acc(i, L). \qquad (1.14)$$

The hard part now is to compute for a fixed i the values $d'(i, L)$ when the values $d(i-1, L)$ are given. In the simple algorithm, we compute $n!$ values in order to compute the minimum in (1.13). Hence all the $n!$ ways of reordering the list state are considered to compute just a single value. This has to be done for every list state L.

Pietrzak computes all $n!$ values $d'(i, L)$ at the same time. The algorithm is based on the fact that a reordering of a list state breaks down to applying a series of at most $\binom{n}{2}$ paid exchanges to the list state.

Let L' be a list state that minimizes the expression on the right hand side of (1.13) and let $L' = L_0, L_1, \ldots, L_k = L$ be the sequence of list states we obtain when reordering the list state from L' to L using single paid exchanges. It is easy to see that

$$d'(i, L_{j+1}) = d'(i, L_j) + 1 \qquad \forall\ 0 \le j < k.$$

This allows to compute $d'(i, L)$ as follows. Let $d'_{(k)}(i, L)$ be the value of the right hand side of (1.13) when we consider for L' only list states that we can obtain from L by at most k exchanges. Define $N(L)$ to be the set of list states that can be obtained from list state L by applying at most one paid exchange. Using

$$d'_{(0)}(i, L) := d(i-1, L)$$

$$d'_{(k)}(i, L) := \min_{L' \in N(L)} \big(d'_{(k-1)}(i, L') + trans(L, L')\big)$$

we can compute $d'_{(\binom{n}{2})}(i, L) = d'(i, L)$ for all L in time n^3. This concludes the description of the algorithm.

Since we prove in Chapter 4 that computing OPT is $\mathcal{NP}$-hard [9], a polynomial algorithm does not exist unless $\mathcal{P} = \mathcal{NP}$.

1.5 Other Models

Besides the full cost and the partial cost model, various other models have been considered. The P^d model is a generalization of the standard model where no free exchanges are allowed and each paid exchange costs d units, whereas the definition of the access costs remains unchanged.

Reingold, Westbrook, and Sleator [29] show that their COUNTER$(k, \{k-1\})$ algorithms are

$$\max(1 + \frac{k+1}{2d}, 1 + \frac{1}{k}(2d + \frac{k+1}{2}))$$

competitive. These algorithms maintain a randomly initialized counter modulo k for each item which is increased on every request to its item. Whenever a counter reaches $k-1$, its item is moved to the front. For each d, there exists a k such that COUNTER$(k, \{k-1\})$ is c-competitive for $c \leq 2.75$. The best competitive ratio is $(5 + \sqrt{17})/2 \approx 2.28$ for $d \to \infty$.

Sleator and Tarjan also analyze models where accessing an item at position i costs $f(i)$ units [31].

There are many results concerning average case analysis for list update algorithms. Here the request sequences are generated by a probability distribution. The requests are independent and the probability for the next request to be to x_i is p_i. The performance is usually compared to the optimal static algorithm STAT. This algorithm maintains the items in non-decreasing order by the probabilities p_i. Rivest [30] showed that there exists a constant b such that

$$E\big[\text{FREQUENCY COUNT}(\sigma)\big] = E\big[\text{STAT}(\sigma)\big] + b.$$

However, b has to be chosen very large. Chung, Hajela, and Seymour [16] proved a similar result for MTF:

$$E\big[\mathsf{MTF}(\sigma)\big] \leq \frac{\pi}{2} E\big[\mathsf{STAT}(\sigma)\big] + b$$

for a much smaller constant b. Gonnet, Munro, and Suwanda [18] proved that this ratio is tight for MTF. Recently, Albers and Mitzenmacher [4] showed

$$E\big[\mathsf{TS}(\sigma)\big] \leq \frac{3}{2} E\big[\mathsf{OPT}(\sigma)\big] + b.$$

This result is stronger since OPT performs much better than STAT. Remember that this ratio only holds for sequences generated by a probability distribution as described above.

Chapter 2

A Lower Bound for the Partial Cost Model

2.1 Introduction

In this chapter, we show a lower bound of 1.50115 for the partial cost model. This improves the trivial lower bound of 1.5 presented in the previous chapter. While the improvement over the previous bound is tiny, the significance of the result lies in the fact that the new value is strictly larger than 1.5. This number is important because it is a tight bound for two-item lists. Previously, many researchers believed that there is a 1.5-competitive algorithm for arbitrarily long lists. Our result shows that this is not possible in the partial cost model and indicates that there is no such algorithm in the full cost model either.

In order to show that no algorithm can be strictly c-competitive, one usually gives a probability distribution over the set of finite request sequences Σ for which one can prove that any algorithm A has expected cost

$$E\big[\mathsf{A}(\sigma)\big] \geq c \cdot E\big[\mathsf{OPT}(\sigma)\big]. \tag{2.1}$$

The expectation is taken over the request sequences chosen in the probability distribution. Inequality (2.1) makes sure that for a given algo-

rithm A, one can always find a request sequence for which

$$\mathsf{A}(\sigma) \geq c \cdot \mathsf{OPT}(\sigma).$$

Because of Yao's theorem [34], we only have to check against all deterministic algorithms.

Theorem 2.1 (Yao) *If there is a probability distribution on request sequences so that (2.1) holds for any deterministic online algorithm A, then (2.1) holds also for any randomized algorithm A.*

In order to show a lower bound for non-strict competitive algorithms, one has to give a whole family of probability distributions, one for each value of b, such that

$$E\big[\mathsf{A}(\sigma)\big] \geq c \cdot E\big[\mathsf{OPT}(\sigma)\big] + b \tag{2.2}$$

holds. In general, the expected optimal offline cost of the request sequences considered will have to grow with the value of b.

Our construction uses a *game tree* where alternately the adversary generates a request and the online algorithm serves it. The adversary is oblivious. That is, he is not informed about the action of the online algorithm. So the game tree has *imperfect information* [25]. We consider a finite tree where – after some requests – the ratio of online versus optimal offline cost is the payoff to the adversary. This defines a zero-sum game, which we solve by linear programming. For a game tree that is sufficiently deep, and restricted to a suitable subset of requests so that the tree is not too large in order to stay solvable, this game has a value of more than 1.50115. This shows that any strictly c-competitive online algorithm fulfills $c \geq 1.50115$. In order to derive from this a new lower bound for the competitive ratio c according to (1.3) with a nonzero constant b, one has to generate arbitrarily long request sequences. This can be achieved by composing the game trees repeatedly, as we will show.

A drawback is our assumption of the partial instead of the full cost model. In the latter model, where a request to the ith item in the list incurs cost i, the known lower bound is $1.5 - 5/(n+5)$ for a list with n

items. This result by Teia [32] yields a lower bound for the competitive ratio much below 1.5 when the list is short. In fact, there is a 1.5-competitive algorithm for lists with no more than 13 items, as we will show in Section 2.7. To prove a lower bound above 1.5 for the full cost model, we would have to extend our construction to lists with at least 14 items.

One reason why it is so hard to find lower bounds might be that the game defined by the list update problem does not have a value, meaning that the competitive ratio of the best online algorithm is larger than the best possible lower bound (Remember that lower bounds are specified by a probability distribution on the request sequences). However, this is not true, as we show in Section 2.8.

2.2 Teia's result

Teia [32] has constructed a lower bound of 1.5 for the competitive ratio in the full cost model. Since we will use his ideas for an improved lower bound in the partial cost model, we explain his proof in the simpler partial cost model.

Teia constructs an adversary strategy for which (2.2) holds with $c = 1.5$. The request sequences are generated in *runs* which are repeated indefinitely. The runs are defined by list states. For the first run, the initial list state $S(\emptyset)$ is considered. To obtain the run from the list, we traverse the list from front to back, requesting each item with equal probability either once or three times. If an item is requested three times, it is moved to the front of the list, otherwise it is left in place. This results in a new list, which determines the next run.

It turns out that the optimal offline algorithm on these sequences can be described as follows.

Algorithm 2.2 (`3MTF`) *Move x to the front if and only if the next three requests are all to x.*

To see that `3MTF` is optimal on any of Teia's sequences σ, note that

3MTF is projective on σ since we have

$$S_{xy}^{\mathsf{3MTF}}(\sigma) = S_{xy}^{\mathsf{3MTF}_{xy}}(\sigma_{xy}).$$

Therefore, using (1.8), we obtain

$$\mathsf{3MTF}(\sigma) = \sum_{\{x,y\}\subseteq L} \mathsf{3MTF}_{xy}(\sigma_{xy}). \tag{2.3}$$

Note that Definition 1.3 does not require an algorithm to be online. Furthermore, Equation (1.8) clearly also holds for offline algorithms which are projective.

Because of (1.9), we only have to prove that 3MTF is optimal on two items. This holds because 3MTF pays exactly one unit per run in this case, which also is a lower bound since in each run, there is at least one request to each item.

Concerning the online player, we have

$$E\big[\mathsf{A}(\sigma)\big] \geq \sum_{\{x,y\}\subseteq L} E\big[\mathsf{A}_{xy}(\sigma_{xy})\big]. \tag{2.4}$$

Here, A_{xy} denotes the optimal online algorithm on the projections of Teia's adversary strategy. The explanation for (2.4) is very similar to that of (1.9).

Teia's proof is based on a potential function Φ. Φ_i is the number of inversions between A's and OPT's list state after the ith run ρ. Since both algorithms start in the same list state, it holds $\Phi_0 = 0$. He proves

$$E\big[\mathsf{A}(\rho) + \Phi_i - \Phi_{i-1}\big] \geq 1.5 \cdot \mathsf{OPT}(\rho). \tag{2.5}$$

For sequences σ with k runs, we get

$$E\big[\mathsf{A}(\sigma) + \Phi_k - \Phi_0\big] \geq 1.5 \cdot \mathsf{OPT}(\sigma).$$

Since we have $0 \leq \Phi_k \leq \binom{n}{2}$ for lists with n items, there exists a k for each $\varepsilon > 0$ such that

$$E\big[\mathsf{A}(\sigma)\big] > (1.5 - \varepsilon) \cdot \mathsf{OPT}(\sigma),$$

ρ_{xy}	OPT with $[xy]$	A with $[xy]$, $\Phi_{i-1}=0$							
		x WAIT y WAIT		x MTF y WAIT		x WAIT y MTF		x MTF y MTF	
		t_i	Φ_i	t_i	Φ_i	t_i	Φ_i	t_i	Φ_i
xy	1	1	0	1	0	1	1	1	1
$xxxy$	1	1	0	1	0	1	1	1	1
$xyyy$	1	2	0	2	0	1	0	1	0
$xxxyyy$	1	2	0	2	0	1	0	1	0
$4 \cdot E\big[t_i + \Phi_i - \Phi_{i-1}\big]$		$6+0$		$6+0$		$4+2$		$4+2$	

Table 2.1: *Expected online cost if* $\Phi_{i-1} = 0$

which proves the result.

In order to prove (2.5), let ρ_{xy} be the projection of the ith run to x and y. Because of (2.4) and (2.3) and $\mathsf{OPT}_{xy}(\rho_{xy}) = 1$, we can prove (2.5) by showing that

$$E\big[\mathsf{A}_{xy}(\rho_{xy}) + \Phi_i - \Phi_{i-1}\big] \geq 1.5 \cdot \mathsf{OPT}_{xy}(\rho_{xy}). \tag{2.6}$$

To keep the notation simple, we also use Φ_i to denote the projection of the potential function to x and y here. We have $\Phi_i = 0$ if there is no inversion between x and y, and $\Phi_i = 1$ otherwise.

The Tables 2.1 and 2.2 show (2.6). Because of symmetry, we only consider runs where OPT's list state initially is $[xy]$. There are four cases for ρ_{xy}, all have the same probability to be chosen. Table 2.1 refers to the case $\Phi_{i-1} = 0$. In Table 2.2, we assume $\Phi_{i-1} = 1$. We can assume that A moves the requested item to the front whenever there are two requests to it in a row. This holds because in this case, A knows that there is another request to the same item following. Therefore moving it to the front is always optimal. Thus, there are only four different ways to serve ρ_{xy}. MTF means that the item is moved to the front already at the first request. WAIT moves it at the second request, if there is one. By t_i we denote the online player's cost. Since $\mathsf{OPT}_{xy}(\rho_{xy}) = 1$ in all cases, we obtain (2.6).

ρ_{xy}	OPT with $[xy]$	A with inversion $[yx]$, $\Phi_{i-1}=1$							
		x WAIT y WAIT		x MTF y WAIT		x WAIT y MTF		x MTF y MTF	
		t_i	Φ_i	t_i	Φ_i	t_i	Φ_i	t_i	Φ_i
xy	1	1	1	2	0	1	1	2	1
$xxxy$	1	3	0	2	0	3	1	2	1
$xyyy$	1	1	0	3	0	1	0	2	0
$xxxyyy$	1	4	0	3	0	3	0	2	0
$4 \cdot E[t_i + \Phi_i - \Phi_{i-1}]$		$9-3$		$10-4$		$8-2$		$8-2$	

Table 2.2: *Expected online cost if* $\Phi_{i-1}=1$

2.3 Poset algorithms

Using *partial orders*, one can construct a 1.5-competitive list update algorithm for lists with up to four items [6]. The partial order is initially equal to the linear order of the items in the list. After each request, the partial order is modified as follows, where $x \| y$ means that x and y are incomparable:

partial order before	after request to		
	$z \notin \{x,y\}$	x	y
$x \| y$	$x \| y$	$x < y$	$y < x$
$x < y$	$x < y$	$x < y$	$x \| y$

That is, a request only affects the requested item y in relation to the remaining items. Then y is in front of all items x except if $x < y$ held before, which is changed to $x \| y$. The initial order in the list and the request sequence determine the resulting partial order. Note the similarity to Automaton 1.2. One can generate an arbitrary partial order in this way [6].

The partial order defines a *position*

$$p(x) = |\{y \mid y < x\}| + |\{y \mid y \| x\}|/2$$

for each item x. If the online algorithm that only uses free exchanges

can maintain a distribution on lists so that the expected cost of accessing an item x is equal to $p(x)$, then this algorithm is 1.5-competitive [6]. One can show that then with probability one x is behind all items y with $y < x$, and precedes with probability 1/2 those items y where $x \| y$. Incomparable elements reflect the possibility of a "mistake" of not transposing these items, which should have probability 1/2. For lists with up to four items, one can maintain such a distribution using two lists only. That is, the partial order is represented as the intersection of two linear orders represented by the lists, where each list is updated by moving the requested item suitably to the front, using only free exchanges. The algorithm works by choosing one of these lists at the beginning with probability 1/2 as the actual list and serving it so as to maintain the partial order (with the aid of the separately stored second list).

The partial order approach is very natural for the projection on pairs and when the online algorithm can only use free exchanges. A lower bound above 1.5 must exploit a failure of this algorithm. This is already possible for lists with five items, despite the fact that all five-element partial orders are two-dimensional (representable as the intersection of two linear orders). Namely, let the items be the letters a to e and let the initial list be $[abcde]$, and consider the request sequences

$$\sigma_1 = dbed \qquad \text{and} \qquad \sigma_2 = dbec\,. \tag{2.7}$$

After the first request to d, the partial order states are $d \| a$, $d \| b$, $d \| c$, and $d < e$, and otherwise $a < b < c < e$. Using a free exchange, d can only be moved forward and has to precede c, b, a each with probability 1/2. This is achieved uniquely with the uniform distribution on the two lists $[abcde]$ and $[dabce]$ (this, as well as the following, holds even though distributions on more than two lists are allowed). The next request to b induces $b < d$, so b must be moved in front of d in the list $[dabce]$, where b already passes a, which yields the unique uniform distribution on $[abcde]$ and $[bdace]$. The next request to e entails that e is incomparable with all other items. It can be handled deterministically in exactly two ways (or by a random choice between these two ways): Either e is moved to the front in $[bdace]$, yielding the two lists $[abcde]$

and $[ebdac]$ with equal probability, or e is moved to the front in $[abcde]$, yielding the two lists $[eabcd]$ and $[bdace]$ with equal probability. If the two lists are $[abcde]$ and $[ebdac]$, the algorithm must disagree with the partial order after the request to d as in σ_1, since then d must precede both a and e in both lists (so d is moved to the front in both lists) but then incorrectly passes b where only $b \| d$ should hold. Similarly, for the two lists $[eabcd]$ and $[bdace]$ the request to c as in σ_2 moves c in front of e and d in both lists, so that it passes a, violating $a \| c$. Thus, either σ_1 or σ_2 in (2.7) causes the poset-based algorithm to fail, which otherwise achieves a competitive ratio of 1.5. These sequences will be used with certain probabilities in our lower bound construction.

2.4 Game trees with imperfect information

As we have seen in Chapter 1, the list update problem can be phrased as a zero-sum game between two players, the adversary and the online algorithm (or *online player*). A lower bound for strictly competitive algorithms can be shown by giving a finite adversary strategy for which (2.1) holds.

In order to deal with finite games, we assume a finite set $S \subseteq \Sigma$ of request sequences (for example all of a given bounded length), which represent the pure strategies of the adversary. These can be *mixed* by randomization. There exist only a finite number N of possible ways of deterministically serving these request sequences in S. These deterministic online algorithms can also be chosen randomly by suitable probabilities p_j for $1 \leq j \leq N$. In this context of finitely many request sequences, an arbitrary constant b in (1.3) is not reasonable, so we look at strict competitiveness. To be strictly c-competitive against the adversary strategies in S, it must hold for all σ in S that

$$\sum_{j=1}^{N} p_j \, \mathsf{A}_j(\sigma) \leq c \cdot \mathsf{OPT}(\sigma) \,, \tag{2.8}$$

where $\mathsf{A}_j(\sigma)$ is the cost incurred by the jth online algorithm and $\mathsf{OPT}(\sigma)$ is the optimal offline cost for serving σ. We can disregard the trivial se-

quences σ with $\mathsf{OPT}(\sigma) = 0$ that consist only of requests to the first item in the list. In this case (2.8) is equivalent to

$$\sum_{j=1}^{N} p_j \frac{\mathsf{A}_j(\sigma)}{\mathsf{OPT}(\sigma)} \leq c\,. \tag{2.9}$$

The terms $\mathsf{A}_j(\sigma)/\mathsf{OPT}(\sigma)$ in (2.9), for $1 \leq j \leq N$ and $\sigma \in S$, can be treated as a payoff to the adversary in a zero-sum game matrix with rows σ and columns j. Correspondingly, a lower bound d for the strict competitive ratio of list update algorithms is an expected competitive ratio [15] resulting from a distribution on request sequences. This distribution is a mixed strategy of the adversary with probabilities q_σ for σ in S so that for all online strategies $j = 1, \ldots, N$

$$\sum_{\sigma \in S} q_\sigma \frac{\mathsf{A}_j(\sigma)}{\mathsf{OPT}(\sigma)} \geq d\,. \tag{2.10}$$

Note that the bounds in (2.9) hold only for the strategies in S, whereas the lower bounds in (2.10) hold in general.

In the finite case, the minimax theorem for zero-sum games [34] asserts that there are mixed strategies for both players and reals c and d so that (2.9) and (2.10) hold with $d = c$. Then c is the "value" of the game and the optimal strict competitive ratio for the chosen finite approximation of the list update problem. Note that it depends on the admitted length of request sequences. Due to the complicated implicit definition and large size of the game matrix, the best known bounds for c and d in (2.9) and (2.10) that hold irrespective of the length of the request sequences do not coincide.

The number of request sequences is exponential in the length of the sequences. The online player has an even larger number of strategies since that player's actions are conditional on the observed requests. This is best described by a *game tree*. At each nonterminal node of the tree, a player makes a move corresponding to an outgoing edge. The game starts at the root of the tree where the adversary chooses the first request. Then, the online player moves with actions corresponding

to the possible reorderings of the list after the request. There are $n!$ actions corresponding to all possible reorderings. (Later, we will see that most of them need not be considered.) The players continue to move alternatingly until the last request and the reaction by the online player. Each leaf of the tree defines a sequence σ and an online cost $\mathsf{A}(\sigma)$ (depending on the online actions leading to that leaf), with payoff $\mathsf{A}(\sigma)/\mathsf{OPT}(\sigma)$ to the adversary.

The restricted information of the adversary in this game tree is modeled by *information sets* [25]. Here, an information set is a set of nodes where the adversary is to move and which are preceded by the same previous moves of the adversary himself. Hence, the nodes in the set differ only by the preceding moves of the online player, which the adversary cannot observe. An action of the adversary is assigned to each information set (rather than an individual node) and is by definition the same action for every node in that set. Hence, the probability for choosing an item, say a, as a next request must be the same in all nodes of the information set. On the other hand, the online player is fully informed about past requests, so his information sets are singletons. Figure 2.1 shows the initial part of the game tree for a list with three items for the first and second request by the adversary, and the first online response, here restricted to free exchanges only.

A pure *strategy* in a game tree assigns a move to every information set of a player, except for those that are unreachable due to an earlier choice of that player. Here, the online player has information sets (like in Figure 2.1) where each combination of moves defines a different strategy. This induces an *exponential* growth of the number of strategies in the *size of the tree*. The strategic approach using a game matrix as in (2.9) and (2.10) becomes therefore computationally intractable even if the game tree is still of moderate size. Instead, we have used a recent method [33, 24] which allows to solve a game tree with a "sequence form" game matrix and corresponding linear program that has the *same* size as the game tree.

Using game trees, a first approach to finding a randomized strategy for the adversary is the following. Consider a list with five items, the minimum number where a competitive ratio above 1.5 is possible. Fix a

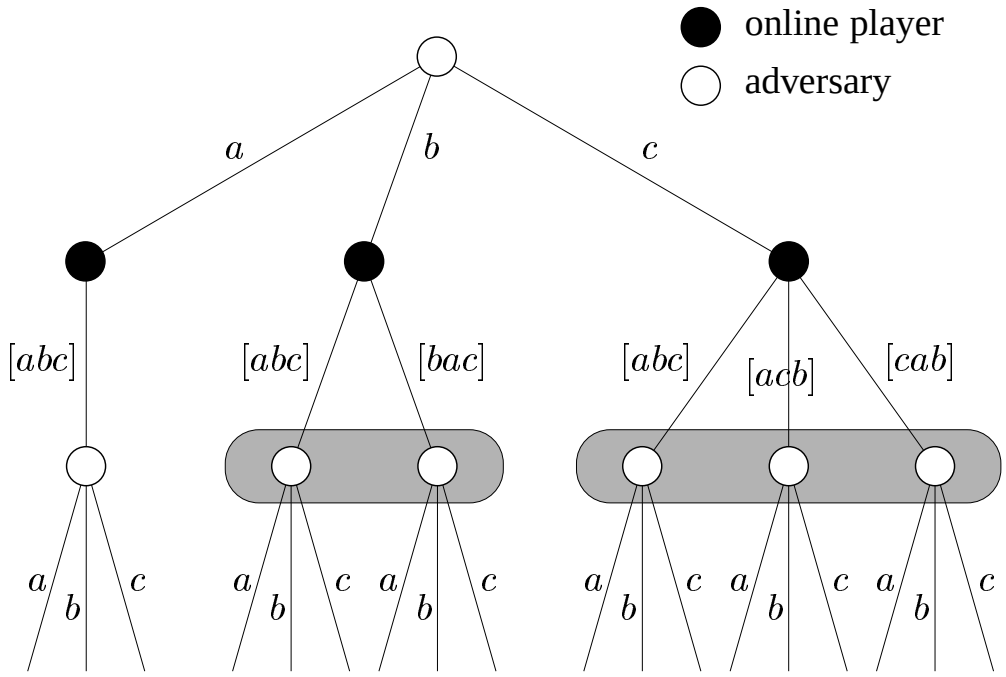

Figure 2.1: *Game tree with information sets.*

maximum length m of request sequences, and generate the game tree for requests up to that length. At each leaf, the payoff to the adversary is the quotient of online and offline cost for serving that sequence. Then convert the game tree to a linear program, and compute optimal strategies with an LP solver (we used CPLEX).

However, this straightforward method does not lead to a strict competitiveness above 1.5, for two reasons. First, "mistakes" of an algorithm manifest themselves only later as actual costs. As an example, if A moves y to the front on the first request of $\sigma = yxxx$, we really need the requests to x to make A pay for this mistake. So there is little hope for an improved lower bound using short request sequences. Secondly, even if only short sequences are considered, the online player has $n!$ responses to every move of the adversary, so that the game tree grows so fast that the LP becomes computationally infeasible already for very small values of m.

The first problem is overcome by adding the number of *inversions* of the online list, denoted by Φ_i in the tables 2.1 and 2.2 above, to the

payoff at each leaf. This yields a strict competitive ratio greater than 1.5 for rather short sequences. The inversions are converted into actual costs by attaching a well structured subgame to each leaf of the game tree that generates requests sequences similar to Teia's lower bound construction. The next section describes the details.

The second problem, the extremely rapid growth of the game tree, is avoided as follows. First, we limit the possible moves of the online player by allowing only paid exchanges of a special form, so-called *subset transfers* [28]. A subset transfer chooses some items in front of the requested item x and puts them in the same order directly behind x (e.g. $[a\underline{b}c\underline{de}xfg] \rightarrow [acx\underline{bde}fg]$). Afterwards, the adversary's strategy computed against this "weak" online player is tested against *all* deterministic strategies of the online player, which can be done quickly by dynamic programming. It turns out that the lower bound still holds, that is, the "strong" online player who may use arbitrary paid exchanges cannot profit from its additional power. Remember that using free exchanges does not help the online player since they can be simulated by paid exchanges, as we have seen on page 6.

2.5 The game tree gadgets

We compose a game tree from two types of trees or "gadgets". The first gadget called *FLUP* (for "finite list update problem") has a small, irregular structure. The second gadget called *IC* (for "inversion converter") is regularly structured. An instance of *IC* is appended to each leaf of *FLUP*. Both gadgets come with a randomized strategy for the adversary, which has been computed by linear programming for *FLUP*. One can prove that against this adversary strategy, the best online strategy has an expected strict competitive ratio of at least $1.5 + 1/864$, about 1.50115. To check all possible online strategies, one can use dynamic programming. The *FLUP* game we used is the shortest that we found; larger versions of *FLUP* give higher lower bounds. If we allowed in the *FLUP* game all request sequences of length at most k, the limit of the game value for $k \rightarrow \infty$ would be the value of the list update game.

requests	offline list	probability	OPT
$d.$	$[dabce]$	$552/1728$	3
$dbe.$	$[eabcd]$	$168/1728$	8
$dbe.$	$[ebacd]$	$56/1728$	8
$dbec.$	$[abcde]$	$360/1728$	10
$dbec.$	$[acebd]$	$60/1728$	10
$dbec.$	$[cebda]$	$300/1728$	10
$dbed.$	$[bdace]$	$29/1728$	9
$dbed.$	$[bdeac]$	$145/1728$	9
$dbed.$	$[deabc]$	$58/1728$	9

Table 2.3: *The adversary strategy*

This follows from Section 2.8.

Both gadgets assume a particular state of the offline list, which is a parameter that determines the adversary strategy. Furthermore, at the root of *FLUP* (which is the beginning of the entire game), it is assumed that both online and offline list are in the same state, say $[abcde]$. Then the adversary strategy for *FLUP* generates only the request sequences d, dbe, $dbec$, and $dbed$ with positive probability, which are the sequences in (2.8) or a prefix thereof. After the responses of the online player to one of these request sequences, the *FLUP* tree terminates in a leaf with a particular status of the online list and of the offline list, where the latter is *also chosen by the adversary*, independently of the online list. For the request sequence d, that offline list is $[dabce]$, that is, the offline algorithm has moved d to the front. If the *FLUP* game terminates after the request sequence dbe, the adversary makes an additional *internal* choice, unobserved by the online player, between the offline lists $[eabcd]$ and $[ebacd]$. In the first case, the offline player brought e to the front but left d and b in their place, in the second, b was also moved to the front. Similar choices are performed between the offline lists for the request sequences $dbec$ and $dbed$.

The specific probabilities for these choices of the adversary in *FLUP* are shown in Table 2.3. The last column denotes the cost for the opti-

mal offline algorithm. The *FLUP* tree starts with d as the first request, followed by the possible responses of the online player. Next, the adversary exits with probability $552/1728$, without a request, to the leaf with offline list $[dabce]$, and with complementary probability requests item b, which is followed by the online move, and so on.

Each leaf of the *FLUP* tree is the root of an *IC* gadget which generates requests (similar to the runs in Teia's construction, see below), depending on the offline list. The number of inversions of the online list relative to this offline list is denoted by Φ. The purpose of the *IC* gadget is to convert these Φ inversions into actual costs. Any request sequence generated by the *IC* gadget can be treated with the same minimal offline cost v, here $v = 30$. Thereby, the online algorithm makes mistakes relative to the offline algorithm, so that the online cost in *IC* is at least $1.5v + \Phi$.

Since adding the *IC* gadgets leads to a game far too large to compute its value, we consider instead the game consisting only of the *FLUP* tree with the following payoffs at its leafs. Let A be the cost the online player has to pay on its path from the root to the leaf. Then the payoff to the adversary at this leaf is

$$D = \frac{\mathsf{A} + \Phi + 1.5v}{\mathsf{OPT} + v}\,. \tag{2.11}$$

Note that the expected payoff to the adversary in this small game is at most as large as in the one with the *IC*s. Therefore, the value of the small game is the desired lower bound.

The probabilities in table 2.3 have been computed by linear programming. One can show that any online strategy, as represented in the *FLUP* tree, has an expected strict competitive ratio of at least $1.5 + \frac{1}{864}$, or about 1.50115.

At a leaf of the *FLUP* gadget, the adversary reveals his list state to the online player and he charges D. This he can do since, as we will see later, there exists a strategy for the adversary which depends only on the adversary's list state which makes the online player pay at least $1.5v + \Phi$ in the expectation, whereas the adversary pays only v. Hence at the leaf, the adversary can indeed guarantee payoff D.

The fact that the strategy does not depend on the online list is crucial, since otherwise the adversary would not be oblivious any more. Revealing the adversary's list to the online player is allowed since it merely weakens the position of the adversary: Any online strategy without this extra information can also be used when the online player is informed about the adversary's internal choice, so then the online player cannot be worse.

The offline list assigned to a leaf of the *FLUP* gadget is part of an optimal offline treatment (computed similar to [28]) for the entire request sequence. However, that list may even be part of a suboptimal offline treatment, which suffices for showing a lower bound since it merely increases the denominator in (2.10). Some of the offline costs in table 2.3 can only be realized with *paid exchanges* by the offline algorithm. For example, the requests $dbec$ are served with cost 10 yielding the offline list $[bcdea]$ by initial paid exchanges that move a to the end of the list. With free exchanges, this can only be achieved by moving every requested item in front of a, which would result in higher costs.

In the remainder of this section, we describe the *IC* gadget. Its purpose is to convert the inversions at the end of the *FLUP* game to real costs while maintaining the lower bound of at least 1.5. At the same time, these inversions are removed so that both the online list and the offline list are in the same order after serving the *IC*.

The *IC* extends the construction by Teia [32] described in Section 2.2. Let T_k be the sequence that requests the first k items of the current offline list in ascending order, requesting each item with probability 1/2 either once or three times. Assume that the offline algorithm treats T_k by moving an item that is requested three times to the front at the first request, while leaving any other item in place, which is optimal. The triply requested items, in reverse order, are then the first items of the new offline list, followed by the remaining items in the order they had before. Then T_n is a run as used in Teia's construction for a list with n items. The random request sequence generated there can be written as T_n^w, that is, a w-fold repetition of T_n, where w goes to infinity. Note that the offline list and hence the order of the requests *changes* from one run T_n to the next, so T_n^2, for example, is *not* a repetition of two

identical sequences. We still have (2.3) for sequences consisting of these runs. The optimal offline treatment of T_k is still the same as in the special case of T_n and costs $\binom{k}{2}$ units.

Next we show

$$E[\mathsf{A}(T_k) - \Phi_{i-1} + \Phi_i] \geq 1.5\,\mathsf{OPT}(T_k)\,, \tag{2.12}$$

where Φ_{i-1} and Φ_i denote the inversions between the list states of the two players before and after the run. All we have to do is to generalize (2.6). Previously, ρ_{xy} was one of the four sequences of Table 2.1. In the general case, ρ_{xy} can also be x, xxx or the empty sequence.

In all three cases, we have $\mathsf{OPT}_{xy}(\rho_{xy}) = 0$. Additionally, we have $E[\mathsf{A}(\rho_{xy}) - \Phi_{i-1}] \geq 0$, because the online algorithm incurs cost at least one if $\Phi_{i-1} = 1$. Hence,

$$E[\mathsf{A}(\rho_{xy}) - \Phi_{i-1} + \Phi_i] \geq 1.5\,\mathsf{OPT}(\rho_{xy}) + E[\Phi_i] \tag{2.13}$$

holds. In order to prove (2.6), we can simply omit the additional term on the right hand side.

As in (2.5) above, (2.12) can be extended to concatenations of sequences T_k. Let us first consider the randomly generated sequence defined by the four runs

$$IC' := T_4 \quad T_3 \quad T_2 \quad T_1, \tag{2.14}$$

which by the preceding considerations fulfills

$$E[\mathsf{A}(IC')] \geq 1.5\,\mathsf{OPT}(IC') + \Phi_0 - E[\Phi_4] \tag{2.15}$$

A more refined analysis shows

$$E[\mathsf{A}(IC')] \geq 1.5\,\mathsf{OPT}(IC') + \Phi_0. \tag{2.16}$$

Namely, if we consider the projection of IC' to any pair of items, the last run is one where only one item is requested, hence (2.13) applies. Therefore we obtain

$$E[\mathsf{A}(IC') - \Phi_0 + \Phi_4] \geq 1.5\,\mathsf{OPT}(IC') + E[\Phi_4]$$

which proves (2.16). Hence, *IC*′ serves as an inversion converter with $v = 10$.

However, the inversion converter we will use is

$$IC := T_4^2 \quad T_3^2 \quad T_2^2 \quad T_1^2, \tag{2.17}$$

because this one allows to prove

$$E[\mathsf{A}(IC)] \geq 1.5\,\mathsf{OPT}(IC) + \Phi_0 + E[\Phi_8]. \tag{2.18}$$

Note that if we project *IC* to any pair of items, the last two runs will be x or xxx. Hence, the projection ends with at least two requests to x in a row. A simple case analysis shows that for the last two runs together, one can prove

$$E[\mathsf{A}(\rho_{xy}) - \Phi_{i-2} + \Phi_i] \geq 1.5\,\mathsf{OPT}(\rho_{xy}) + 2 \cdot E[\Phi_i] \tag{2.19}$$

Here, we have $\rho_{xy} \in \{xx, xxxx, xxxxxx\}$, and Φ_{i-2} and Φ_i count the inversions before and after the last two runs. Again, we have $\mathsf{OPT}(\rho_{xy}) = 0$. The following case analysis proves (2.19) and therefore also (2.18).

Φ_{i-2}	Φ_i	$E[\mathsf{A}(\rho_{xy}) - \Phi_{i-2} + \Phi_i]$	$2 \cdot E[\Phi_i]$
0	0	≥ 0	0
0	1	≥ 2	2
1	0	≥ 1	0
1	1	≥ 2	2

Using *IC*, we obtain a lower bound above 1.5 for the competitive ratio c in (2.2) for any additive constant b by arguing as follows. If we assume that the online player uses MTF to serve the inversion converter, we have $\Phi_8 = 0$ at the leaves of the *IC*. Since the list states of the two players coincide, we can add new *FLUP* games at the leaves of the inversion converter. Doing this recursively, we can generate request sequences of arbitrary length and offline cost. Hence, our bound holds for any constant b in (2.2).

Indeed, there is no use for the online player to have inversions at the end of the inversion converter since by (2.18), he pays exactly the amount he would have to pay for creating them as a first step in the next *FLUP* game.

2.6 Free exchange model

In the above construction, the value of the lower bound does not depend on whether the online player may use paid exchanges or not, but the adversary's strategy does use paid exchanges. So it seems that the online player cannot gain additional power from paid exchanges. This raises the conjecture that by restricting both players to free exchanges only, the list update problem might still have an optimal competitive ratio of 1.5. However, this is false. There is a randomized adversary strategy where the offline algorithm uses only free exchanges which cannot be served better than with a competitive ratio of $1.5 + 1/3644$.

In the previous case, the pure strategies of the adversary consisted of request sequences with at most four requests plus an inversion converter. This is short enough such that one can allow all request sequences of length four for the adversary. In the optimal randomized strategy computed by the linear program, most of them are chosen with probability zero.

In the free exchange model, one needs longer request sequences to find lower bounds larger than 1.5. Since the number of request sequences grows exponentially, the brute force method is not tractable anymore. However, there is a way to generate small sets of pure strategies with serve as candidates for our method.

If one restricts the online player to a small number of random bits and restricts the length of the request sequences, the problem can be viewed as a finite two-person zero-sum game with full information. Let r be the number of random bits allowed to the online player. This can be modeled as follows. Like in the case of the poset algorithms, the state of the online player consists of 2^r list states. These list states can be observed by the adversary. As a first move of the online player, he uses the r random bits to chose one of the 2^r list states uniformly at random as the one he will use to update the list accordingly. This choice is not revealed to the adversary.

A request is served by changing all 2^r list states by the online player. The game trees end with a move of the adversary, where he chooses

requests	offline list	probability	OPT
$d.$	$[abcde]$	$3312/7288$	3
$dd.$	$[dabce]$	$1587/7288$	3
$ddbe.$	$[ebdac]$	$783/7288$	9
$ddbea.$	$[aedbc]$	$120/7288$	10
$ddbea.$	$[dabce]$	$165/7288$	10
$ddbea.$	$[daebc]$	$75/7288$	10
$ddbeac.$	$[dcabe]$	$396/7288$	13
$ddbeacb.$	$[abced]$	$238/7288$	14
$ddbeacb.$	$[bdcae]$	$102/7288$	14
$ddbeacbe.$	$[abced]$	$111/7288$	17
$ddbeacbe.$	$[ebcda]$	$111/7288$	17
$ddbeace.$	$[aebcd]$	$35/7288$	15
$ddbeace.$	$[aedbc]$	$35/7288$	15
$ddbeace.$	$[ecdab]$	$70/7288$	15
$ddbeaceb.$	$[aebdc]$	$74/7288$	17
$ddbeaceb.$	$[bcaed]$	$74/7288$	17

Table 2.4: *Adversary strategy in the free exchange model*

a list state which serves as inversion converter. The payoff is defined by (2.11), where A and Φ are the expected total online cost and the expected number of inversions taken over the r random bits.

Setting $r := 2$, one finds a set of 14 pure strategies of length at most eight which allow to prove the desired bound.

In the case with paid exchanges, we had a simple argument why no online player would ever preserve inversions after the *IC*. Namely we showed that for every inversion the online player kept after the *IC* he would have to pay an extra cost. Instead of preserving inversions in the *IC*, he could w.l.o.g. create them just after the *IC*, spending the same amount.

Changing our models slightly, we can use a similar argument in the case with free exchanges. If we repeat the T_k in (2.17) exactly t times instead of three times, preserving an inversion now costs $t - 2$ units instead of

requests	offline list	probability	OPT
$d.$	$[dabce]$	$1541/3377$	3
$dbe.$	$[eabcd]$	$924/3377$	8
$dbec.$	$[abcde]$	$135/3377$	10
$dbec.$	$[bacde]$	$405/3377$	10
$dbeccb.$	$[cbade]$	$372/3377$	11

Table 2.5: *Adversary strategy in the MTF model*

only one. Note that since there are only free exchanges allowed, it is not possible to create a new inversion between item x and y once the last request to x or y has been served. Of course the offline cost for an *IC* grows with t. By allowing the online player to use paid exchanges that cost $t-2$ units per exchange just after leaving the *IC* and before entering the next *FLUP* game, we only strengthen the online player as there is no obligation to use them. In the new model, we can again assume that no inversions are kept in the *IC*, as the online player can perform these special paid exchanges before the *FLUP* game.

But this time, we really have to check whether the online player can take advantage of the paid exchanges or not. This just needs a simple extension of our dynamic programming approach. If he can, the value of t has to be increased.

For the adversary strategy in Table 2.4, the best online algorithm achieves a competitive factor of $1.5+\frac{1}{3644}$ using the original *IC*.

The same can be done in the model where elements either stay at their current position or are moved to the front of the list. The strategy presented in Table 2.5 also uses the regular inversion converter and proves a lower bound of $1.5+\frac{2}{3377}$.

2.7 Full cost model

The method we presented here cannot be applied to the full cost model because one would have to use lists with at least fourteen items. The

reason for this is the existence of a 1.5-competitive algorithm COMB13 for lists with up to 13 items in the full cost model. Like COMB, COMB13 is a combination of TS and BIT. But this time, BIT is chosen with probability $5/6$, whereas we choose TS with probability $1/6$.

Clearly, COMB13 is projective and we can prove its competitive ratio along the lines of the proof of Theorem 1.10.

In the full cost model, accessing the item at position i costs i units instead of $i-1$ units in the partial cost model. This holds for both COMB13 and OPT. In order to apply projective analysis, the additional unit has to be equally distributed among all pairs involved in the current request. There are exactly $n-1$ pairs involved in each request, namely those which contain the currently requested item. If the first item in the pair list is requested, we charge $\frac{1}{n-1}$ units. For the second item, $1+\frac{1}{n-1}$ units are charged. Doing so, the access cost for the item at position i really becomes

$$(i-1)\cdot\big(1+\frac{1}{n-1}\big)+(n-i)\cdot\big(\frac{1}{n-1}\big)=i,$$

as we require in the full cost model.

As in the proof for COMB, we show that the amortized cost projected to any pair of items is bounded by 1.5 times OPT's cost. Figure 2.2 shows the access cost of COMB13 on pairs of items for $n=13$.

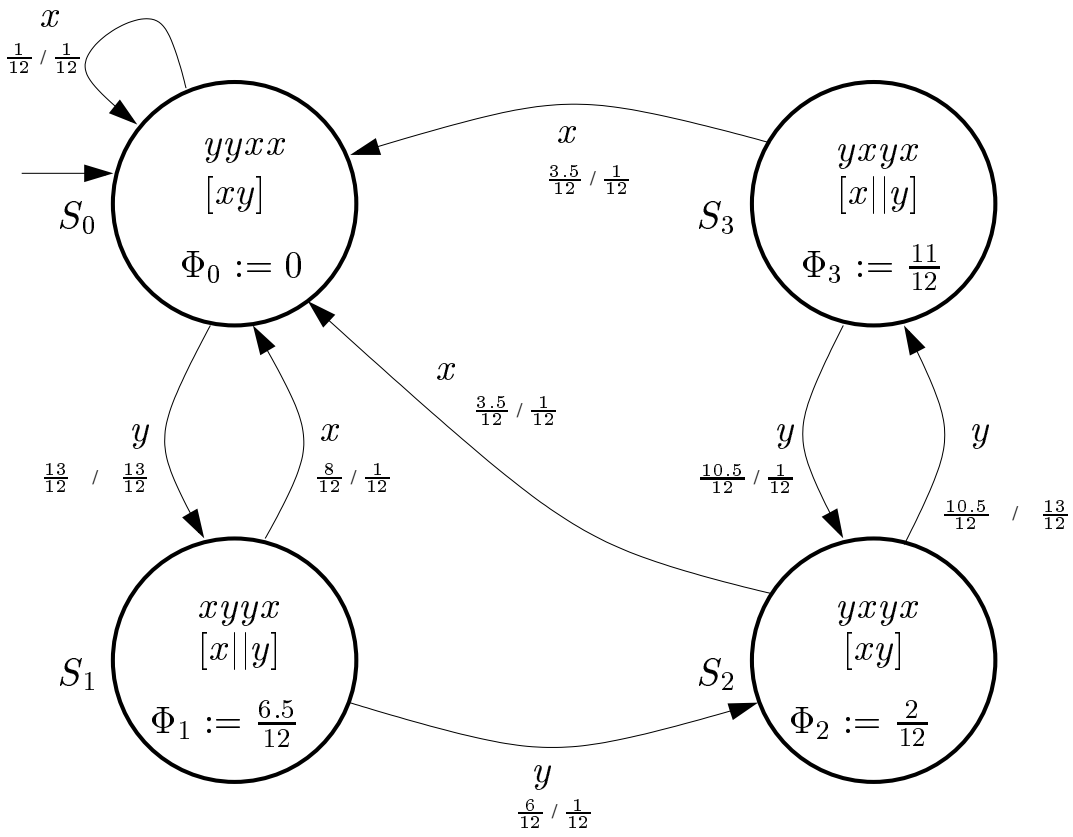

Figure 2.2: *Automaton proving that* `COMB13` *is* 1.5*-competitive on lists with up to 13 items in the full cost model.*

2.8 The list update game has a value

Think of the list update problem as a two-person zero-sum game. The pure strategies of the adversary are the finite request sequences. Those of the online player are the deterministic algorithms. Let us first assume a fixed number of items n and a fixed constant b.

Let c be the best competitive ratio an algorithm can attain,

$$c = \inf_{\mathsf{A}} \sup_{\sigma} \frac{\mathsf{A}(\sigma)}{\mathsf{OPT}(\sigma) + b}. \tag{2.20}$$

We disregard the request sequences with $\mathsf{OPT}(\sigma) = 0$ in order to have a well defined payoff in all cases. Note that in (2.20), both A and σ denote randomized strategies. On the other hand, d shall be the value of the best randomized adversary strategy, for simplicity also called sigma,

$$d = \sup_{\sigma} \inf_{\mathsf{A}} \frac{\mathsf{A}(\sigma)}{\mathsf{OPT}(\sigma) + b}.$$

Theorem 2.3 *For n and b fixed, we have $d = c$.*

To get an intuitive understanding, the reason why this theorem holds is that the online algorithm can force the adversary to somehow restart the whole game after a bounded number of requests. In this way, even very large request sequences can be broken into subsequences of bounded length for which the minimax theorem applies.

Let us first deal with strictly competitive algorithms in the full cost model.

Proof. Assume $d < c$ and choose k such that

$$k > \frac{\binom{n}{2}(c+1) + dn}{c - d}$$

or equivalently

$$\frac{d \cdot (k+n) + \binom{n}{2}}{k - \binom{n}{2}} < c. \tag{2.21}$$

Let A_k be an online algorithm that is optimally strictly competitive on request sequences with offline cost at most $k+n$. Clearly, A_k is strictly d-competitive on such sequences, since the lower bounds increase with the length of the request sequences one allows as adversary strategies.

A_k can be extended to an algorithm for arbitrary long request sequences as follows. It behaves regularly until $k \leq \mathsf{OPT}(\sigma) \leq k+n$. At this time, we reset the game to the initial state. This means that A_k moves to the initial list state which costs at most $\binom{n}{2}$ units. Concerning the adversary, he is allowed to execute a special update operation which moves its list state to the initial one as well. The cost for this operation is negative. His costs get reduced by $\binom{n}{2}$. We can assume that the adversary always performs this reset operation since the saved $\binom{n}{2}$ units allow him to move to any desired list state at the beginning of the next phase with net cost at most zero for the two operations. Hence we are again in the initial state of the algorithm and we can serve the next phase.

In each complete phase, A_k pays at most $d \cdot (k+n) + \binom{n}{2}$ units, whereas the adversary pays at least $k - \binom{n}{2}$ units. If the last phase is not completed (it does not end with a reset operation), A_k pays at most $d \cdot k'$ units, whereas the adversary pays k' units. Hence, the cost ratio is strictly smaller than c in every phase. Therefore A_k is strictly c'-competitive for $c' < c$, which is a contradiction.

Note that by introducing the reset operations, we bounded the adversary's cost from below. Hence the ratio of $\mathsf{A}_k(\sigma)$ and $\mathsf{OPT}(\sigma)$ is not larger. □

For the partial cost model, this proof does not work because there exist infinitely many request sequences σ with $\mathsf{OPT}(\sigma) \leq k$ and therefore it is not clear whether there exists a d-competitive algorithm on these sequences. However, the following lemma proves that there is at least a $d/(1-\varepsilon)$-competitive algorithm for any small $\varepsilon > 0$. This allows to apply the above proof, since we can choose ε such that

$$d < d/(1-\varepsilon) < c.$$

Lemma 2.4 *Let d be the best lower bound for strictly competitive algorithms on list with n items. Then there exists, for every k and any*

$0 < \varepsilon < 0.5$, a $d\frac{1+\varepsilon}{1-\varepsilon}$-competitive algorithm if the adversary is restricted to request sequences with $\mathsf{OPT}(\sigma) \leq k$.

Proof. Let us restrict the adversary to request sequences of length at most w with

$$w = \left\lceil \frac{d \cdot k}{\varepsilon} \right\rceil + 1.$$

Certainly there is a strictly d-competitive algorithm A' on sequences of length at most w. The value of w is chosen such that all request sequences σ for which $\mathsf{OPT}(\sigma) \leq k$ holds and A' pays at least ε for every request satisfy $|\sigma| < w$. To see this, all we have to prove is that there is no such σ with $|\sigma| = w$. Note that A' would pay at least $w\varepsilon > dk$ for such a sequence. This contradicts the fact that A' is strictly d-competitive on sequences of length w.

Let us now prove the lemma by designing an algorithm A which is strictly $d\frac{1+\varepsilon}{1-\varepsilon}$-competitive on request sequences with $\mathsf{OPT}(\sigma) \leq k$.

The algorithm A internally runs algorithm A'. In general, the two algorithms will be in the same list state. In the remainder of this proof, all the costs are expected costs.

Upon each request to an item x, it first checks whether the cost to serve x in A' is larger than ε. If this is the case, x is fed to A' and we perform in A the same paid exchanges as in A' and access x.

If serving x costs less than ε in A', algorithm A enters a so-called ε-phase by moving x to the front of the list in A, which costs at most ε units. As long as there are requests to x, item x is kept at the front of the list. Hence, the access cost for all these requests are zero. None of these requests is fed to A', though. On the first request to an item y different from x, we leave the ε-phase by moving x back to the place where it was before the ε-phase. This costs at most ε units. Note that for the request to y, both A and A' will have to pay at least $1 - \varepsilon$ units. Hence if $0 \leq \varepsilon < 0.5$, this request will be processed regularly.

Let σ be a request sequence with $\mathsf{OPT}(\sigma) \leq k$. Let further σ' be the subsequence which was fed to A'. Since we have $\mathsf{OPT}(\sigma') \leq \mathsf{OPT}(\sigma) \leq k$ and A' payed at least ε units for every request, we have $|\sigma'| < w$ and therefore A' is d-competitive on σ'.

Note that we have $A(\sigma) \leq A'(\sigma') + r2\varepsilon$, where r is the number of times that A enters an ε-phase. We claim than

$$\mathsf{A}(\sigma) \leq \frac{1+\varepsilon}{1-\varepsilon} \cdot \mathsf{A}'(\sigma').$$

The idea is that for any cost (1-eps) spent by A', algorithm A is allowed to spend (1-eps+2eps). Note that we can charge the 2ε for every ε-phase to the first request after the previous ε-phase, since A' pays at least $1-\varepsilon$ for these requests. Note that if σ starts with an ε-phase, this phase is free for both algorithms since in this case, the requested item is already at the front of the list. Otherwise, the first request of σ is charged for the first ε-phase, since its cost is at least one unit. Hence we conclude

$$\mathsf{A}(\sigma) \leq \frac{1+\varepsilon}{1-\varepsilon} \cdot \mathsf{A}'(\sigma') \leq d\frac{1+\varepsilon}{1-\varepsilon} \cdot \mathsf{OPT}(\sigma') \leq d\frac{1+\varepsilon}{1-\varepsilon} \cdot \mathsf{OPT}(\sigma).$$

□

The case where $b > 0$ is similar. Namely, for any ε, we can choose k large enough such that (2.21) holds for $(d+\varepsilon)$ instead of d and

$$\mathsf{A}_k(\sigma) \leq (d+\varepsilon) \cdot \mathsf{OPT}(\sigma)$$

for all sequences with $k \leq \mathsf{OPT}(\sigma) \leq k+n$. In this way, we are sure that all the completed phases are fine. For the last phase, we use the constant b.

What we are actually interested to prove is

$$\lim_{n\to\infty} \lim_{b\to\infty} c(n,b) = \lim_{n\to\infty} \lim_{b\to\infty} d(n,b).$$

First of all, the two limits to exists. This follows from

$$1 \leq d(n,b) \leq c(n,b) \leq 1.6$$

and the fact that $c(n,b)$ and $d(n,b)$ decrease monotonically for growing b and increase monotonically for growing n. Furthermore, Theorem 2.3 makes sure that the limits indeed have the same value.

Chapter 3

Optimal Bounds for Projective Algorithms

3.1 Introduction

Although (1.7) is a necessary and sufficient condition for projective algorithms, it gives not much insight of how projective algorithms can be constructed and whether there are better algorithms than the known families of algorithms. Consequently, obtaining a lower bound for projective algorithms seems to be very hard.

In this chapter, we present a simple characterization of all projective algorithms. The crucial part of the characterization was already presented in the introduction of this thesis in terms of the critical request algorithms.

A key observation for our result is to look at algorithms in a more static way. The classical definition of algorithms as `BIT` or `TS` is in terms of how the current list state changes upon a new request. Our approach tries to understand how $S(\sigma)$ is determined by σ and the initial list state, without considering the evolution of the list states. More specifically, we ask how $S(\sigma)$ changes if the requests of σ are permuted.

While the critical request algorithms cover already all "efficient" pro-

jective algorithms, we have to extend their definition in order to really cover the whole class of projective algorithms. Namely, two functions for every item are needed. The functions F_x are basically the critical request functions known from the introduction. Note that if the relative ordering of two items in $S(\sigma)$ is defined by critical requests of σ, it is not possible to have a pair of items whose relative order remains unchanged on shuffling the requests in σ. Hence, FREQUENCY COUNT for example, although projective, cannot be described as a critical request algorithm. We extend the critical request algorithms by additional functions C_x which group all items into so-called containers. The containers are totally ordered. The ordering of items in different containers is then determined by the ordering of the containers, while items within a container are generally ordered by their critical requests.

The following theorem gives the desired characterization. For a request sequence σ with i requests to x, let $\sigma_x = x^i$ (the i-fold repetition of x) denote the subsequence consisting only of the requests to item x.

Theorem 3.1 *A is a deterministic projective algorithm for a set L of list items, if and only if there exists an ordered set $\mathcal{W} = \mathcal{W}^+ \dot\cup \mathcal{W}^-$ and two functions*

$$\begin{array}{lcllll} C & : & L \times \mathbb{N}_0 & \mapsto & \mathcal{W}, & C(x,i) \neq C(y,0), \quad \forall (x,i) \neq (y,0), \\ F & : & L \times \mathbb{N} & \mapsto & \mathbb{N}, & F(x,i) \leq i, \qquad\quad \forall (x,i) \end{array}$$

with the following properties: given any two items $x \neq y$, and any request sequence σ such that $\sigma_x = x^i, \sigma_y = y^j$, x is in front of y in the online list after A has served σ if one of the following three conditions holds.

(a) $C(x,i) < C(y,j)$, *or*

(b1) $C(x,i) \in \mathcal{W}^+$ *and there exists a pair (z,k), $z \neq x,y$ such that $C(x,i) = C(y,j) = C(z,k)$, and the $F(x,i)$-th request to x appears* **after** *the $F(y,j)$-th request to y in σ, or*

(b2) $C(x,i) \in \mathcal{W}^-$ *and there exists a pair (z,k), $z \neq x,y$ such that $C(x,i) = C(y,j) = C(z,k)$, and the $F(x,i)$-th request to x appears* **before** *the $F(y,j)$-th request to y in σ.*

If none of these conditions hold, both relative orderings of x and y are allowed.

We will also write $C(x, i)$ as $C_x(i)$ and $F(x, i)$ as $F_x(i)$. All pairs (x, i) which map to the same value $w \in \mathcal{W}$ under C define an equivalence class which we call a *container* and identify with w. We say that x is in container w with respect to σ if $C(x, |\sigma_x|) = w$. As a shortcut, we write $x^i \in w$, $i = |\sigma_x|$. By $[x^i]$ we denote the container w for which $C(x, i) = w$. Initially, each item x is in a container $C(x, 0)$ of its own, whose position in the order represents the initial list state.

If at least three items are in some container with respect to σ, the relative order of any two of them (x and y, say) after serving σ is determined by the order of their *critical requests* $F_x(i)$ and $F_y(j)$ in σ. In case of $C_x(i) \in \mathcal{W}^+$, we have the item in front whose critical request is more recent. In this case, we have a *standard container*, otherwise a *nonstandard* one.

The theorem does not completely specify the behavior of A in case there are containers with only two items. In this case, there is no restriction on the order of the two items, except the obvious condition that this order does not depend on requests to other items. In particular, any algorithm over a two-item list is projective, in which case the theorem holds with suitable C and arbitrary F.

Let us illustrate this theorem for a few projective algorithms over the set of items $L = \{x_1, \ldots, x_n\}$, with initial list state $[x_1, \ldots, x_n]$. In this case, MTF uses $\mathcal{W} = \mathcal{W}^+ = \{0, \ldots, n\}$ and

$$\begin{aligned} C(x_k, i) &= \begin{cases} k, & \text{if } i = 0 \\ 0, & \text{otherwise} \end{cases} \\ F(x_k, i) &= i. \end{aligned}$$

Thus, MTF moves all items into a common container after their first request. This container is a standard container, so x is in front of y in the online list if and only if x was requested more recently than y. Using $\mathcal{W} = \mathcal{W}^- = \{0, \ldots, n\}$ instead would result in the MOVE-TO-BACK algorithm, which is not competitive. We will see later that

no competitive algorithm will use nonstandard containers with positive probability.

The algorithm TIMESTAMP moves all items into a common standard container after their second request; within that container, items are ordered by recency of their second-to-last request. This behavior can be obtained by using $\mathcal{W} = \mathcal{W}^+ = \{0, \ldots, 2n\}$ and

$$\begin{aligned} C(x_k, i) &= \begin{cases} 2k, & \text{if } i = 0 \\ 2k-1, & \text{if } i = 1 \\ 0, & \text{otherwise} \end{cases} \\ F(x_k, i) &= \max(1, i-1). \end{aligned}$$

The randomized algorithm BIT tosses a coin for each item x do decide whether x will be moved to the front after an even number of requests to x (and stay in place after an odd number of requests), or vice versa. Thus, BIT uses $\mathcal{W} = \mathcal{W}^+ = \{0, \ldots, 2n\}$ and for each k randomly decides between the two pairs of functions

$$\begin{aligned} C_1(x_k, i) &= \begin{cases} 2k, & \text{if } i = 0 \\ 2k-1, & \text{if } i = 1 \\ 0, & \text{otherwise} \end{cases} \\ F_1(x_k, i) &= \max(1, 2\lfloor i/2 \rfloor) \end{aligned}$$

and

$$\begin{aligned} C_2(x_k, i) &= \begin{cases} 2k, & \text{if } i = 0 \\ 0, & \text{otherwise} \end{cases} \\ F_2(x_k, i) &= 2\lceil i/2 \rceil - 1. \end{aligned}$$

Finally, we consider the algorithm FREQUENCY COUNT which maintains the items sorted according to decreasing number of past requests; two items which have been requested equally often are ordered by recency of their last request, like in MTF. This corresponds to the choices of $\mathcal{W} = \mathcal{W}^+ = \mathbb{Z}$ and

$$\begin{aligned} C(x_k, i) &= \begin{cases} k, & \text{if } i = 0 \\ -i, & \text{otherwise} \end{cases} \\ f(x_k, i) &= i. \end{aligned}$$

FREQUENCY COUNT is not competitive; in fact, we will prove that no competitive algorithm maintains more than one container in the long run with positive probability.

Using the characterization of Theorem 3.1, we will derive the lower bound for the competitive ratio of any projective algorithm.

Theorem 3.2 *For any $\varepsilon > 0$, any $b \in \mathbb{R}$ and any projective algorithm A, there exists a finite sequence λ such that*

$$E[A(\lambda)] \geq (1.6 - \varepsilon)OPT(\lambda) + b.$$

These results are significant in two respects. First, they show that the successful approach of combining existing projective algorithms to obtain improved ones has reached its limit with the development of the COMB algorithm. New and better algorithms (if they exist) have to be non-projective, and must derive from new, yet to be discovered, design principles.

Second, the characterization of projective algorithms is a step forward in understanding the structural properties of list update algorithms. Under this characterization, the largest and so far most significant class of algorithms appears in a new, unified way.

Projective algorithms have a natural generalization, where we demand the relative order of any k-tuple of list items to depend only on the requests to these k items. It turns out that for lists with more than k items, only projective algorithms satisfy this condition. This follows from the fact that e.g. for $k = 3$,

$$\begin{aligned} S^A_{xyz}(\sigma) &= S^A_{xyz}(\sigma_{xyz}) \qquad \text{and} \\ S^A_{xyw}(\sigma) &= S^A_{xyw}(\sigma_{xyw}) \end{aligned}$$

imply that the list state $S_{xy}(\sigma)$ depends only on σ_{xy}, because it must be independent of w and z.

We define a *randomized* online algorithm as projective if it is a discrete probability distribution over deterministic projective algorithms. A less restrictive definition is conceivable, but would not allow us to prove

the lower bound for projective algorithms that we intend and that we think is useful. Namely, one could call a randomized online list update algorithm projective if serving any request sequence σ induces a distribution on list states $S_{xy}(\sigma)$ that only depends on σ_{xy}. Under these weaker requirements, one can indeed find 1.5-competitive algorithms for lists with few items. For the case of two items, Algorithm 1.12 trivially is projective. Furthermore, the Poset algorithms described in Section 2.3 are also projective in this generalized sence. Unfortunanely, they are defined only for lists with up to four items.

Theorem 3.1, discussed further and proved in the following sections, characterizes the deterministic projective algorithms in a way that makes their projective behavior transparent, and unifies many known algorithms. By our above assumption that considers a randomized projective algorithm as a probability distribution over deterministic ones, we will be able to use this characterization in the lower bound proof of Theorem 3.2 later.

An open problem is to extend the lower bound to the full cost model, even though this model is not very natural in connection with projective algorithms. This would require request sequences over arbitrarily many items, and it is not clear whether an approach similar to the one given here can work.

3.2 Containers

Consider a deterministic projective algorithm A over a set L of list items with fixed initial list state; our intended characterization in the form of Theorem 3.1 addresses the relative order of two items $x \neq y$ in the online list after a sequence σ with $\sigma_x = x^i, \sigma_y = y^j$ has been served. An easy case occurs if this order only depends on i and j, but not on the pattern in which the requests appear in σ. For example, FREQUENCY COUNT has x in front of y whenever $i > j$. This leads us to the following

Definition 3.3 *Let $x, y \in L, x \neq y$ and $i, j \geq 0$. We define*

$$x^i \to_A y^j \quad :\Leftrightarrow \quad \exists \sigma : \sigma_x = x^i, \sigma_y = y^j, S^A_{xy} = [xy].$$

Observation 3.4 *If x,y and z are distinct and i, j, $k \geq 0$, then $x^i \to_A y^j$ and $y^j \to_A z^k$ implies $x^i \to_A z^k$.*

To see this, consider some σ with $\sigma_x = x^i$, $\sigma_y = y^j$ and $S^A_{xy} = [xy]$. Without affecting σ_{xy}, we can insert k requests to z into σ, and because of $y^j \to_A z^k$, this can be done in such a way that $S^A_{yz}(\sigma) = [yz]$ Because A is projective, we then have $S^A(\sigma) = [xyz]$ which proves $x^i \to_A z^k$.

For distinct items, $\to_A$ is transitive by Observation 3.4. In general, we obtain a reflexive and transitive relation from $\to_A$ as follows.

Definition 3.5 *Let $\overset{*}{\to}_A$ be the reflexive and transitive closure of the relation $\to_A$. Then*

$$x^i \sim_A y^j \quad :\Leftrightarrow \quad x^i \overset{*}{\to}_A y^j \text{ and } y^j \overset{*}{\to}_A x^i.$$

defines an equivalence relation whose equivalence classes are the containers *determined by A.*

By $[x^i]$ we denote the container x^i belongs to. A container w is *grown* if there exist distinct items x, y, and z and i, j, $k \geq 0$ such that $w = [x^i] = [y^j] = [z^k]$.

Lemma 3.6 *If $[x^i]$ is grown, $y^j \in [x^i]$ and $x \neq y$, then* $x^i \to_A y^j$.

Proof. Because $[x^i]$ is grown, there is a projection $z^k \in [x^i]$ with z distinct from x and y. Hence, $x^i \overset{*}{\to}_A z^k$ and $z^k \overset{*}{\to}_A y^j$. Therefore there exists a chain

$$x^i = q_1^{k_1} \to_A q_2^{k_2} \to_A \cdots \to_A q_{n-1}^{k_{n-1}} \to_A q_n^{k_n} = y^j$$

with $z = q_\ell$ for some $\ell \neq 1, n$. By successively removing all intermediate elements of the chain, we will derive $x^i \to_A y^j$. Consider the first

index s such that $\{q_1, \ldots, q_s\}$ contains three distinct elements. Because of $q_i \neq q_{i+1}$ for $i = 1, \ldots, n-1$, the items q_s, q_{s-1} and q_{s-2} must be distinct. By Observation 3.4, $q_{s-1}^{k_{s-1}}$ can be removed from the chain without affecting its validity, because $q_{s-2}^{k_{s-2}} \rightarrow_{\mathsf{A}} q_s^{k_s}$.

Only for $s = 3$, this might result in a chain containing projections to only two distinct items. In the case $n = 3$, this is okay since we then obtain $x^i \rightarrow_{\mathsf{A}} y^j$, which we were looking for. In the case $n > 3$, it is easy to check that q_{s-1}, q_s and q_{s+1} are distinct, and the removal of $q_s^{k_s}$ yields a shorter chain, again containing projections to three distinct elements.

Continuing in this fashion, we finally arrive at the desired chain $x^i \rightarrow_{\mathsf{A}} y^j$. □

Corollary 3.7 *If w is a grown container and $x^i, y^j \in w$, then there exist sequences σ, σ' with $\sigma_x = \sigma'_x = x^i$, $\sigma_y = \sigma'_y = y^i$ and*

$$\begin{aligned} S_{xy}^{\mathsf{A}}(\sigma) &= [xy], \\ S_{xy}^{\mathsf{A}}(\sigma') &= [yx]. \end{aligned}$$

This means that, for projections in the same container, the order of the respective elements depends on the pattern in which the requests to them occur in σ. For projections in different containers, the order can be derived from a suitable order on the containers.

Lemma 3.8 *There exists a total order $<_{\mathsf{A}}$ on the containers such that $w_1 <_{\mathsf{A}} w_2$ for distinct w_1, w_2 implies that for all pairs $x^i \in w_1$, $y^j \in w_2$, $x \neq y$: $x^i \rightarrow_{\mathsf{A}} y^j$ and not $y^j \rightarrow_{\mathsf{A}} x^i$.*

Proof. The relation

$$w_1 <_{\mathsf{A}} w_2 :\Leftrightarrow \exists x^i \in w_1, y^j \in w_2 : x^i \overset{*}{\rightarrow}_{\mathsf{A}} y^j$$

is a partial order which can be extended to a total order; it follows that for any pair $[x^i] \neq [y^j]$ with $x \neq y$, $[x^i] \rightarrow_{\mathsf{A}} [y^j]$ if and only if $[x^i] <_{\mathsf{A}} [y^j]$.

To see that $<_{\mathsf{A}}$ is indeed a partial order, we have to show that it is reflexive, transitive and antisymmetric. The first two hold because $\stackrel{*}{\rightarrow}_{\mathsf{A}}$ has these two properties as well. The last property holds because of the definition of the containers in Definition 3.5. □

Concerning the proof of Theorem 3.1, we have defined the container functions C and have dealt with case (a) of the theorem.

3.3 Critical Requests

In this section, we deal with the case where $C(x, |\sigma_x|) = C(y, |\sigma_y|)$. The interesting case here is when this container is grown. Under this condition, the relative order of x and y in the online list after serving σ can be characterized in terms of *critical requests*.

Theorem 3.9 *Let A be a deterministic projective algorithm over a set of items L. Then there exists a function*

$$F \;:\; L \times \mathbb{N}^+ \;\mapsto\; \mathbb{N}^+, \qquad F(x,i) \leq i, \quad \forall (x,i)$$

such that for all grown containers w exactly one of the following conditions holds.

*(b1) For all pairs x^i, y^j with $w = [x^i] = [y^j]$, and all σ such that $\sigma_x = x^i, \sigma_y = y^j$, $S^{\mathsf{A}}_{xy}(\sigma) = [xy]$ if and only if the $F(x,i)$-th request to x happens **after** the $F(y,j)$-th request to y in σ.*

*(b2) For all pairs x^i, y^j with $w = [x^i] = [y^j]$, and all σ such that $\sigma_x = x^i, \sigma_y = y^j$, $S^{\mathsf{A}}_{xy}(\sigma) = [xy]$ if and only if the $F(x,i)$-th request to x happens **before** the $F(y,j)$-th request to y in σ.*

In case (*b1*), we say that w is a *standard* container, in case (*b2*) we have a *nonstandard* one. This also yields the partition of the set of containers $\mathcal{W}$ into $\mathcal{W}^+$ and $\mathcal{W}^-$ that we have stipulated in Theorem 3.1.

As a simple illustration, observe that the algorithm MTF uses $F(x,i) = i$ for all x and satisfies condition (*b1*).

Proof. Let σ be a request sequence with $\sigma_x = x^i, \sigma_y = y^j, i, j > 0$ and $|\sigma| = i + j$ such that $[x^i] = [y^j]$, and $[x^i]$ is grown.

We label each request to an item with its position in the unary projection to that item (e.g. the fifth request to x will be labeled $x_{(5)}$). Then σ can be considered as a permutation of labeled requests. Because of Corollary 3.7, there exists a permutation σ' of σ such that $S^{\mathsf{A}}_{xy}(\sigma) \neq S^{\mathsf{A}}_{xy}(\sigma')$. This means that, we can as well assume that in σ, we have a consecutive pair of items $x_{(q)}, y_{(l)}$, such that $S^{\mathsf{A}}_{xy}(\sigma) \neq S^{\mathsf{A}}_{xy}(\sigma')$, where σ' arises from σ by transposing $x_{(q)}$ and $y_{(l)}$.

This behavior does not change if we add $k > 0$ requests to $z \neq x, y$ to σ. We choose z and k such that $[z^k] = [x^i]$. Because $[x^i]$ is grown, such a pair must exist.

We add these requests to σ such that $x_{(q)}$ and $y_{(l)}$ stay consecutive and such that there exists a consecutive pair of requests, say $x_{(q')}$ and $z_{(m)}$ such that transposing this pair changes $S_{xz}(\sigma)$. Again because of Corollary 3.7, this must be possible.

We claim that $q = q'$. To see this, assume $q \neq q'$. Then we can, if necessary, transpose any of the two pairs $x_{(q)}, y_{(l)}$ and $x_{(q')}, z_{(m)}$ in σ such that $S(\sigma) = [yxz]$. Let $\tilde{\sigma}$ be σ with the two pairs $(x_{(q)}, y_{(l)})$ and $(x_{(q')}, z_{(m)})$ transposed. By projectivity, we have $S(\tilde{\sigma}) = [zxy]$. Hence $S_{yz}(\sigma) \neq S_{yz}(\tilde{\sigma})$, although $\sigma_{yz} = \tilde{\sigma}_{yz}$. This contradicts the projectivity of A.

In particular, there is a unique value of q such that $x_{(q)}$ participates in any transposition that reorders x and z in the list. By symmetry, this uniqueness also holds for the pair x and y. Because the value of q is the same in both cases and by projectivity, it only depends on σ_x. Therefore it is a function of $i = |\sigma_x|$. We call $x_{(q)}$ the critical request of x and set $F_x(i) = F(x, i) = q$.

By a symmetric argument, the request $x_{(\ell)}$ defines the unique critical request for y, and $F_y(j) = F(y, j) := \ell$.

To see that the relative order of two items in the list must change whenever the two critical requests are transposed in the request sequence, think of a request sequence σ on two items x and y where the critical requests of x and y are consecutive. Let σ' be the sequence obtained by

swapping the critical requests. Now assume that this swap does not alter the list state. Then we can obtain any permutation of the request sequence by successively transposing consecutive requests, starting from either σ or σ', without ever transposing the critical requests. Thus, the relative order of x and y would be the same for all request sequences. This contradicts our assumption that $x^i \sim_{\mathsf{A}} y^j$.

We still need to argue that the items in w are ordered either according to case (*b1*) or (*b2*).

For this, consider a request sequence σ over an n-item list such that $S(\sigma) = [x_1 x_2 \dots x_n]$. Let p_i be the position of x_i's critical request in σ. If we do not have

$$\begin{aligned} p_1 > p_2 > \cdots > p_n \quad & \text{(case (}b1\text{)) or} \\ p_1 < p_2 < \cdots < p_n \quad & \text{(case (}b2\text{)),} \end{aligned}$$

we must have an index i such that either

$$p_i < p_{i+1} > p_{i+2} \qquad \text{or} \qquad p_i > p_{i+1} < p_{i+2}.$$

In both cases, we can manipulate σ such that the critical requests of x_i and x_{i+2} change their order, but both keep their relative order w.r.t. the critical request of x_{i+1}. In the list obtained after serving σ, items x_i and x_{i+2} change their relative order under this manipulation, while they keep their relative order with respect to x_{i+1}. This is impossible. □

The assumption of grown containers in the preceding theorem is crucial. If $x \neq y$ and $y^j \in [x^i]$, where $[x^i]$ is a non-grown container, then x and y are adjacent in $S^{\mathsf{A}}(\sigma)$, for any σ with $\sigma_x = x^i$ and $\sigma_y = y^j$. This holds because $[x^i]$ does not contain a projection to a third element. In this case, the projective algorithm is free to choose any order of x and y which only depends on σ_{xy}, without violating projectivity. In particular, the algorithm is not forced to operate according to critical requests.

Together with the results of the previous section, we have now proved Theorem 3.1.

3.4 The Lower Bound

In this section, we use the characterization of projective algorithms from Theorem 3.1 to prove that no such algorithm is better than 1.6-competitive. Intuitively, it is clear that a good algorithm will maintain only one container in the long run (which must be a standard container), and it will have the critical request close to the last request for each item. We prove this intuition in the next section; for the time being, we restrict our attention to algorithms which satisfy these conditions.

Definition 3.10 *For a given integer $M > 0$, a deterministic projective algorithm is called* M-regular*, if*

(i) $C_x(i) = C_y(j)$ *(and this container is a standard container) for all items x, y and all $i, j \geq M$, and*

(ii) $f_x(i) := i - F_x(i) < M$ *for all items x and all $i \geq 1$.*

A randomized algorithm is M-regular if it is a probability distribution over deterministic M-regular algorithms.

Except FREQUENCY COUNT (which is not competitive), all the algorithms discussed at the end of the introduction are M-regular with $M \in \{1, 2\}$.

Given any $\varepsilon > 0$ and b, we will show that there is a probability distribution π on a finite set Λ of request sequences so that

$$\sum_{\lambda \in \Lambda} \pi(\lambda) \frac{\mathsf{A}(\lambda)}{\mathsf{OPT}(\lambda) + b} \geq 1.6 - \varepsilon, \tag{3.1}$$

for any deterministic M-regular algorithm A. Then *Yao's theorem* [34] asserts that also any randomized M-regular algorithm has competitive ratio $1.6 - \varepsilon$ or larger. This holds for any $\varepsilon > 0$, so the competitive ratio is at least 1.6. This is achieved by COMB and therefore 1.6 is a tight bound for the competitive ratio of M-regular algorithms.

All $\lambda \in \Lambda$ will consist of only two items x and y. In what follows, let $\hat{M} > M$ and $\hat{M} \geq 3$ and let

$$\phi := x^{\hat{M}}\, yxyx^{\hat{M}}\, yx^{\hat{M}}\, y^{\hat{M}}\, xyxy^{\hat{M}}\, xy^{\hat{M}}\, x^{\hat{M}}\, y^{\hat{M}}. \tag{3.2}$$

ϕ consists of eight *blocks*, each of which ends in $x^{\hat{M}}$ or $y^{\hat{M}}$. Let K and T be positive integers and

$$H := |\phi|/2 = 4\hat{M} + 4. \tag{3.3}$$

Then the set of sequences in (3.1) is given by

$$\Lambda = \Lambda(K,T) := \{x^{\hat{M}+t} y^{\hat{M}+h} \phi^K \mid 0 \leq h < H, 0 \leq t < TH\}, \tag{3.4}$$

where any λ in Λ is chosen with equal probability $1/H^2T$ by π.

OPT pays exactly ten units for each repetition of ϕ (which always starts in offline list state $[yx]$). Assuming that also the initial list state is $[yx]$, all sequences in Λ have offline cost $10K + 2$. This and the fact that $\pi(\lambda)$ for $\lambda \in \Lambda$ is constant allows us to conclude (3.1) once we can prove

Lemma 3.11

$$\sum_{\lambda \in \Lambda} A(\lambda) \geq 16KTH^2 - o(KTH^2).$$

Namely, we then obtain

$$\begin{aligned}
\sum_{\lambda \in \Lambda} \pi(\lambda) \frac{\mathsf{A}(\lambda)}{\mathsf{OPT}(\lambda) + b} &= \frac{\sum_{\lambda \in \Lambda} \mathsf{A}(\lambda)}{\sum_{\lambda \in \Lambda} (\mathsf{OPT}(\lambda) + b)} \\
&\geq \frac{16KTH^2 - o(KTH^2)}{(10K + 2 + b)H^2T} \\
&= 1.6 - \frac{1.6(b+2)}{10K + 2 + b} - \frac{o(KTH^2)}{(10K + 2 + b)H^2T} \\
&\geq 1.6 - \varepsilon,
\end{aligned}$$

for $K, T, \hat{M}$ large enough.

In the rest of this section we show that (3.4) yields Lemma 3.11. For this, we distribute the total online cost among *states* assumed by sequences $\lambda \in \Lambda$.

Definition 3.12

(i) $\lambda \in \Lambda$ *assumes state* (i,j) *if there exists a prefix* $\sigma =: \lambda(i,j)$ *of* λ *with* $\sigma_x = x^i$ *and* $\sigma_y = y^j$. U *denotes the set of states assumed by sequences* $\lambda \in \Lambda$.

(ii) $\lambda \in \Lambda$ switches at (i,j), *if* $\lambda(i,j)$ *contains the initial prefix* $x^{\hat{M}+t}y^{\hat{M}+\ell}$, *and one of the eight blocks of some repetition of* ϕ *starts immediately after* $\lambda(i,j)$. *If the block starts with* x, λ switches to x, *otherwise* λ switches to y.

(iii) (i,j) *is called a* switching *state if some* $\lambda \in \Lambda$ *switches at* (i,j). S *denotes the set of switching states.*

(iv) For $(i,j) \in S$ *and* λ *switching at* (i,j), $\mathsf{A}_\lambda(i,j)$ *denotes the online cost incurred by serving the block of* ϕ *that follows the prefix* $\lambda(i,j)$. $\mathsf{A}(i,j)$ *is the sum of these costs over all* λ *switching at* (i,j).

These definitions allow us to rewrite the total online cost as follows.

$$\begin{aligned}\sum_{\lambda\in\Lambda}\mathsf{A}(\lambda) &= \sum_{(i,j)\in S}\mathsf{A}(i,j) + \sum_{0\le h<H, 0\le t<TH}\mathsf{A}(x^{\hat{M}+t}y^{\hat{M}+h}) \\ &> \sum_{(i,j)\in S}\mathsf{A}(i,j).\end{aligned}$$

We see that $\lambda = x^{\hat{M}+t}y^{\hat{M}+h}\phi^K$ switches at (i,j) if and only if

$$\begin{aligned} i &= \hat{M}+t+qH+r, \\ j &= \hat{M}+h+qH+s, \end{aligned} \tag{3.5}$$

for some $q < K$ and

$$(r,s) \in \{(\hat{M},0), (2\hat{M}+1,2), (3\hat{M}+1,3), (4\hat{M}+4, 3\hat{M}+4)\}$$

(switch to y), or

$$(r, s) \in \{(0, 0), (3\hat{M}+1, \hat{M}+3), (3\hat{M}+3, 2\hat{M}+4), (3\hat{M}+4, 3\hat{M}+4)\}$$

(switch to x). For a fixed pair (r, s), the values of h and q (and hence of t) that satisfy these equations are uniquely determined. It follows that at most eight sequences switch at any given state (i, j).

Definition 3.13 *A state $(i, j) \in S$ is called* good *if and only if the following two conditions are satisfied.*

(i) there are exactly eight sequences $\lambda \in \Lambda$ that switch at (i, j), and

(ii) property (i) also holds for the states $(i-1, j)$ and $(i, j-1)$.

G denotes the set of good states.

Then (3.5) further yields

$$\sum_{\lambda \in \Lambda} \mathsf{A}(\lambda) > \sum_{(i,j) \in G} \mathsf{A}(i, j). \tag{3.6}$$

This means, for every good state (i, j) and each of the eight blocks in ϕ, there is exactly one sequence $\lambda \in \Lambda$ such that λ continues with this block after the prefix $\lambda(i, j)$. Moreover, $\mathsf{A}(i, j)$ accounts for the online cost incurred by serving these eight blocks.

We will now prove two claims, which together yield Lemma 3.11 and therefore the lower bound of 1.6.

Claim 3.14 *For every state $(i, j) \in G$, we have $\mathsf{A}(i, j) \geq 16$. The sequences switching to x and y, respectively, both provide eight units.*

Claim 3.15 $|G| \geq KTH^2 - o(KTH^2)$.

Let us prove Claim 3.15 first. From equations (3.5), we see that exactly eight sequences switch at (i, j) if and only if for all eight pairs (r, s),

the solutions q, h, t to system (3.5) satisfy $0 \leq q < K$, $0 \leq h < H$ and $0 \leq t < TH$. Using the facts that $s \leq H, 0 \leq r - s \leq H$, for all (r, s), one proves that a sufficient condition for this is

$$H - 1 \leq j - \hat{M} < KH, \quad j + H \leq i < j + TH - H + 1. \quad (3.7)$$

If both weak inequalities in (3.7) hold as strict inequalities, then (i, j) is guaranteed to be good. Hence there are at least

$$(K - 1)H \cdot (TH - 2H) = KTH^2 - o(KTH^2)$$

good states. This implies the claim.

To prove Claim 3.14, consider a good state (i, j) and the four sequences $\lambda^{(1)}, \ldots, \lambda^{(4)}$ that switch to y in (i, j) (the argument for the sequences switching to x is symmetric). For $\mathsf{A}(i, j)$, we have to count the total online cost incurred by serving the four blocks $y^{\hat{M}}$, $y^{\hat{M}}$, $yx^{\hat{M}}$, and $yxyx^{\hat{M}}$ following the prefixes $\lambda^{(p)}(i, j), p = 1, \ldots, 4$, see Figure 3.1.

$$\begin{array}{lll|llll} \lambda^{(1)}: & \lambda^{(1)}(i,j) & y_{(j+1)} & y_{(j+2)} & \cdots & & \\ \lambda^{(2)}: & \lambda^{(2)}(i,j) & y_{(j+1)} & y_{(j+2)} & y_{(j+3)} & \cdots & \\ \lambda^{(3)}: & \lambda^{(3)}(i,j) & y_{(j+1)} & x_{(i+1)} & x_{(i+2)} & \cdots & \\ \lambda^{(4)}: & \lambda^{(4)}(i,j) & y_{(j+1)} & x_{(i+1)} & y_{(j+2)} & x_{(i+2)} & \cdots \end{array}$$

Figure 3.1: *Blocks in sequences switching from x in (i, j)*

Our goal is to show that A incurs at least eight units of cost by serving the four blocks. This is not always true: a certain choice of critical request values may result in an online cost of only seven units. However, this particular choice will lead to nine units of cost in state $(i-1, j)$, one of which we can "borrow" for $\mathsf{A}(i, j)$. This results in eight amortized units of online cost, for all good states.

Because A is M-regular, we know that x and y are in the same standard container after processing $\lambda^{(p)}(i, j)$ (and also at any later time), for all p. Moreover, the critical request of x is among the preceding $\hat{M}$ requests to x, while the critical request of y is further away. Therefore, four units are to be paid for the requests to $y_{(j+1)}$. At least three more

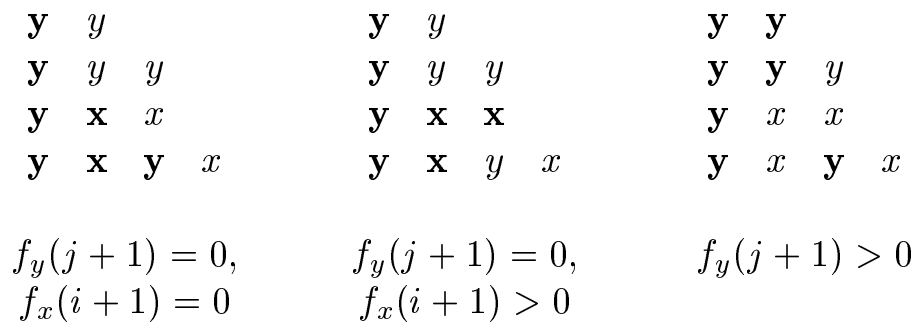

Figure 3.2: *The seven unavoidable cost units*

units are necessary to serve the remaining requests. Figure 3.2 depicts the different cases (requests which incur a cost unit appear in bold). An eighth unit will be spent, unless

$$f_x(i+1) = 0 \qquad \text{and} \qquad f_y(j+2) = 1. \tag{3.8}$$

Namely, $f_y(j+2) \leq 1$ is necessary since otherwise A would have to pay a unit for the request to $y_{(j+3)}$ in $\lambda^{(2)}$. But then we must also have $f_y(j+2) = 1$ and $f_x(i+1) = 0$ to avoid a cost unit for the request to $x_{(i+2)}$ in $\lambda^{(4)}$.

Now we see that for the sequences switching to x in state $(i-1, j)$, two cost units are created by (3.8) in addition to the seven unavoidable units. More specifically, the requests $y_{(j+2)}$ and $y_{(j+3)}$ in the sequence

$$\lambda^{(4)}: \quad \lambda^{(4)}(i-1,j) \;\big|\; x_{(i)} \quad y_{(j+1)} \quad x_{(i+1)} \quad y_{(j+2)} \quad y_{(j+3)} \quad \cdots$$

will cause two cost units which add to the seven unavoidable cost units we spend for these sequences. From the nine cost units in total, we can safely borrow one.

3.5 Irregular Algorithms

Unbounded f–functions

We first show that the lower bound also holds for randomized algorithms even if they use deterministic algorithms which do not satisfy condition (ii) of Definition 3.10 with positive probability. The idea of the proof is to show that by choosing $\hat{M}$ large enough, one can still charge enough to prove the previous lower bound.

In the regular lower bound construction, we have charged 16 cost units for every state (i, j) such that

$$H \leq j - \hat{M} < KH, \quad j + H \leq i < j + TH - H, \tag{3.9}$$

a consequence of (3.7). Let us denote this set of states by G'. As a precondition, we need $f_x(i) < \hat{M}$ to charge for the blocks switching to y and $f_y(j) < \hat{M}$ for those switching to x. In the case of larger f-values, we can still charge

$$\sum_{\lambda \in \Lambda} \mathsf{A}(\lambda) >$$

$$\sum_{(i,j) \in G'} \big(8 \cdot (1 - \mathrm{prob}(f_x(i) \geq \hat{M})) + 8 \cdot (1 - \mathrm{prob}(f_y(j) \geq \hat{M}))\big).$$

The probabilities refer to the probability distribution which defines the randomized algorithm. We can do this because Claim 3.14 already holds if $f_y(j) < \hat{M}$ and $f_x(i) < \hat{M}$. The only case where this is not obvious is the third one in Figure 3.2: in case of $f_x(i+1) \geq \hat{M}$, we might not be able to charge a seventh cost unit. Nevertheless, we will then have at least one cost unit for either $x_{(i+1)}$ or $y_{(j+2)}$. Also, (3.8) ensures that the ninth unit we might need to borrow from another state is actually spent.

Using Claim 3.15 and (3.9), we obtain

$$\sum_{\lambda\in\Lambda}\mathsf{A}(\lambda) > 16KTH^2 - o(KTH^2)$$
$$-\,8\sum_{j=H+\hat{M}}^{KH+\hat{M}-1}\ \sum_{i=j+H}^{j+TH-H-1}\big(\mathrm{prob}(f_x(i)\geq\hat{M})+\mathrm{prob}(f_y(j)\geq\hat{M})\big)$$

Lemma 3.16 *If A is c-competitive for $c \leq 1.6$, then for $i \geq \hat{M}$,*

$$\sum_{\ell=0}^{\hat{M}-1}\mathrm{prob}(f_x(i+\ell)\geq\hat{M}) \leq E(\mathsf{A}(x^i y^{\hat{M}} x^{\hat{M}})) \leq 3c+b, \qquad (3.10)$$

and for $j \geq \hat{M}$,

$$\sum_{\ell=0}^{\hat{M}-1}\mathrm{prob}(f_y(j+\ell)\geq\hat{M}) \leq E(\mathsf{A}(y^j x^{\hat{M}} y^{\hat{M}})) \leq 3c+b. \qquad (3.11)$$

Proof. We only prove (3.10) here, (3.11) is similar. For the first inequality, we only consider the access cost for the last $\hat{M}$ requests to x in the sequence on the right hand side. For each of them, one unit is spent whenever $f_x(i+\ell) \geq \hat{M}$ because then x is behind y. The second inequality holds because A is c-competitive and $\mathsf{OPT}(x^i y^{\hat{M}} x^{\hat{M}}) \leq 3$. □

Using (3.10), we can bound

$$\begin{aligned}
&\sum_{j=H+\hat{M}}^{KH+\hat{M}-1}\ \sum_{i=j+H}^{j+TH-H-1}\mathrm{prob}(f_x(i)\geq\hat{M})\\
&\quad\leq \sum_{j=H+\hat{M}}^{KH+\hat{M}-1}\frac{1}{\hat{M}}\sum_{i=j+H-(\hat{M}-1)}^{j+TH-H-1}\ \sum_{\ell=0}^{\hat{M}-1}\mathrm{prob}(f_x(i+\ell)\geq\hat{M})\\
&\quad\leq (KH-H-1)\frac{1}{\hat{M}}(TH-2H+\hat{M}-2)(2c+b)\\
&\quad= O(KTH^2/\hat{M}) = O(KT\hat{M}) = o(KT\hat{M}^2).
\end{aligned}$$

Inequality (3.11) yields

$$\begin{aligned}
&\sum_{j=H+\hat{M}}^{KH+\hat{M}-1} \sum_{i=j+H}^{j+TH-H-1} \mathrm{prob}(f_y(j) \geq \hat{M}) \\
&\leq \sum_{j=H+\hat{M}-(\hat{M}-1)}^{KH+\hat{M}-1} \frac{1}{\hat{M}}(TH-2H-2) \sum_{\ell=0}^{\hat{M}-1} \mathrm{prob}(f_y(j+\ell) \geq \hat{M}) \\
&\leq (KH-H+\hat{M}-2)\frac{1}{\hat{M}}(TH-2H-2)(3c+b) \\
&= O(KTH^2/\hat{M}) = O(KT\hat{M}) = o(KT\hat{M}^2).
\end{aligned}$$

As in Lemma 3.11, we get

$$\sum_{\lambda\in\Lambda} \mathsf{A}(\lambda) \geq 16KTH^2 - o(KTH^2) - o(KT\hat{M}^2),$$

from which we conclude

$$\sum_{\lambda\in\Lambda} \pi(\lambda)\frac{\mathsf{A}(\lambda)}{\mathsf{OPT}(\lambda)+b} \quad \geq \quad 1.6-\varepsilon,$$

for K, T and $\hat{M}$ large enough.

Several Containers

Proving that our container cannot be a non-standard one is not too hard. In a non-standard container a sequence like $\sigma = xy^i$ would cause unbounded cost for growing i, while the offline cost is a constant. Thus no algorithm can afford to use this kind of container with positive probability.

To show that our lower bound still holds if we allow A to use more than one container, we compare A with an algorithm $\bar{\mathsf{A}}$. We derive $\bar{\mathsf{A}}$ by using A's critical request functions, but ignoring its container structure. Thus in $\bar{\mathsf{A}}$, all items are in a common container after their first request.

As we already proved a lower bound for this kind of algorithms in the previous section, $\bar{\mathsf{A}}$ cannot be more competitive than 1.6.

From the fact that A is supposedly $(1.6-\varepsilon)$–competitive while $\bar{\mathsf{A}}$ is only 1.6–competitive, we derive that there must exist a $\mu > 0$ such that there is a sequence of states (i_k, j_k), $1 \leq k \leq N$, with i_k and j_k increasing strictly monotonically and $\mathrm{prob}(C_x(i_k) \neq C_y(j_k)) \geq \mu$. Remember that A and $\bar{\mathsf{A}}$ can serve a request differently only by using containers.

We will show that this contradicts the competitiveness of A, by exhibiting a family of request sequences on which A cannot be competitive at all.

Let $X_{k,\ell}$ be the indicator variable for the event that $C_x(i_k) \neq C_y(j_\ell)$. If $X_{h,h} = 1$, we get

$$\sum_{1 \leq k', l' \leq N} (X_{k',\ell'} + X_{k',h} + X_{h,\ell'}) \geq N^2 \tag{3.12}$$

This just follows from the fact that $X_{h,h} = 1$, $X_{k',h} = 0$, $X_{h,\ell'} = 0$ imply $X_{k',\ell'} = 1$. Define

$$p_{k,\ell} := E(X_{k,\ell}) = \mathrm{prob}\big(C_x(i_k) \neq C_y(j_\ell)\big).$$

Then we get

$$\begin{aligned} \sum_{1 \leq k', l' \leq N} (p_{k',\ell'} + p_{k',h} + p_{h,\ell'}) &= \\ E[\sum_{1 \leq k', l' \leq N} (X_{k',\ell'} + X_{k',h} + X_{h,\ell'})] &\geq \\ \mathrm{prob}(X_{h,h} = 1) \cdot E[\sum_{1 \leq k', l' \leq N} (X_{k',\ell'} + X_{k',h} + X_{h,\ell'}) | X_{h,h} = 1] &\geq \\ \mu N^2. & \end{aligned}$$

We now sum up for all $X_{h,h}$, $1 \leq h \leq N$ and get

$$\sum_{1 \leq h \leq N} \sum_{1 \leq k', \ell' \leq N} (p_{k',\ell'} + p_{k',h} + p_{h,\ell'}) \geq \mu N^2 \cdot N. \tag{3.13}$$

Observing that each term appears exactly $3N$ times, we find

$$\sum_{1\le k,\ell\le N} p_{k,\ell} \ge \frac{\mu N^3}{3N} \ge \frac{\mu}{3}N^2.$$

Define

$$p^{<}_{k,\ell} := \mathrm{prob}(C_x(i_k) < C_y(j_\ell)),$$
$$p^{>}_{k,\ell} := \mathrm{prob}(C_x(i_k) > C_y(j_\ell)),$$

and assume w.l.o.g. that

$$\sum_{1\le k,\ell\le N} p^{<}_{k,\ell} \ge \frac{\mu}{6}N^2.$$

Then there exists some $t \in \{1,\ldots,N\}$ such that

$$\sum_{\ell=1}^{N} p^{<}_{t,\ell} = \sum_{\ell=1}^{N} \mathrm{prob}\big(C_x(i_t) < C_y(j_\ell)\big) \ge \frac{\mu}{6}N.$$

The request sequence we feed to A is now $\sigma = x^{i_t}y^{j_N+1}$. We have $\mathsf{OPT}(\sigma) \le 2$ but expected online cost at least $\mu/6 \cdot N$, because already the expected cost to serve the $(j_\ell+1)$st request to y is at least

$$p_{t,\ell} = \mathrm{prob}\big(C_x(i_t) < C_y(j_\ell)\big),$$

for all $j \in \{1,\ldots,N\}$. Namely, with this probability, y is at that time in a container behind the one x was moved to in the i_t-th request to x, in which case y incurs online cost of one. By letting N tend to infinity, this shows that A cannot be competitive with constant ratio, which is a contradiction.

Chapter 4

Offline List Update is $\mathcal{NP}$-hard

4.1 Introduction

In this chapter, we will be concerned with the offline version of the list update problem (*OLUP*). Given a request sequence σ and an initial list state L, we would like to compute the minimal offline cost to serve the sequence σ. This value is denoted by $\mathtt{OPT}(L, \sigma)$.

In competitive analysis, the cost of an online algorithm is compared to the cost of the optimal offline algorithm OPT. Understanding OPT might therefore lead to a better understanding of the list update problem itself. Unfortunately, it will turn out in this chapter that the problem of computing $\mathtt{OPT}(L, \sigma)$ is $\mathcal{NP}$-hard. Hence, there is probably not much structure to be understood. Some properties of OPT have already been studied in the past [28, 29, 5, 2].

The significance of this result is increased by the lower bound presented in Chapter 3. Since projective algorithms are analyzed on lists with two items, the structure of OPT is not really an issue here because, as one can see in (1.10), we actually prove the stronger result

$$A(\sigma) \leq c \cdot \overline{\mathtt{OPT}}(\sigma) + b.$$

Thus, it might be necessary to replace $\overline{\texttt{OPT}}$ by better bounds in order to beat COMB.

The currently best algorithm for *OLUP* on lists with n items and request sequences of length m runs in $O(n!n^3m)$ [26] as presented in Section 1.4. From the $\mathcal{NP}$-hardness we can conclude that there cannot be an algorithm which is polynomial in n and m unless $\mathcal{P} = \mathcal{NP}$.

An *OLUP* instance on n items and m requests to these items can be encoded in $\Theta(\log(n) \cdot m)$ bits. But we can assume $m \geq n$. Therefore, an algorithm is still polynomial if its runtime is polynomial in n.

A feasible (but not necessarily optimal) solution for an instance of *OLUP* is here called a *schedule*. Note that there are always optimal schedules which do not involve free exchanges [14]. Therefore a schedule is determined by the sequence of list states $L_1 \ldots L_m$ where L_i denotes the state when the ith request is performed.

As an important part of the proof, we introduce a generalization of *OLUP* called *weighted list update problem* (*WLUP*). Here, the items have a weight that influences access and transposition cost. A version of *WLUP* was considered already in [8], but our definitions and applications are different.

In the proof of the result, we assume the partial cost model. It is easy to obtain the value of $OPT(L, \sigma)$ in the full cost model by adding $|\sigma|$ to the optimal cost in the partial cost model. Therefore, the proof certainly holds for the full cost model as well.

4.2 The Weighted List Update Problem

In this section, we introduce the weighted list update problem (*WLUP*), which generalizes *OLUP* to items with *weights*. These weights have to be non-negative integers. We denote weighted items by capital letters.

An instance of *WLUP* consists of a request sequence σ and an initial list L over a set of weighted items. In general, we denote an instance by the triple (L, σ, W), where w_i is the ith entry of the vector W and denotes the weight of the ith item in L. We denote the optimal algorithm for

this problem by WOPT and the minimal cost for an instance (σ, L, W) by $WOPT(L, \sigma, W)$.

The cost incurred by operating on weighted items is the following. Let the items be X_i, $i = 1 \ldots n$. In order to transpose two items X_i and X_j with weights w_i and w_j respectively, we pay

$$w_i \cdot w_j \tag{4.1}$$

units. The access cost for an item X_i with weight w_i is the following. Let S be the set of items in front of X_i in L_k, then accessing X_i in L_k costs

$$w_i \cdot \sum_{l:X_l \in S} w_l. \tag{4.2}$$

From these definitions it follows that an instance consisting only of items with weight one is identical to an *OLUP* instance. We call the items in an *OLUP* instance *regular* items. Note that if an item has weight zero, it does not cause any cost at all.

Theorem 4.1 *If all weights of a WLUP instance are bounded by a polynomial in the number of items, then there is a polynomial reduction from WLUPto OLUP.*

The reduction is defined by a function f that converts a *WLUP* instance into an *OLUP* instance. Let the *WLUP* instance be (L, σ, W) with items X_i for $i = 1 \ldots n$. The *OLUP* instance will be built by regular items $x_{i,j}$, $i = 1 \ldots n$, $j = 1 \ldots w_i$. We convert (L, σ, W) into an *OLUP* instance by replacing any occurrence of X_i in L and σ by the sequence $x_{i,1} x_{i,2} \ldots x_{i,w_i}$. The theorem follows immediately from the next lemma, since the sum on the left hand side of (4.3) can be computed in polynomial time.

Lemma 4.2 *Let $|\sigma_{X_i}|$ denote the number of occurrences of X_i in σ. Then we have*

$$WOPT(L, \sigma, W) + \sum_{X_i \in L} \binom{w_i}{2} \cdot |\sigma_{X_i}| = OPT(f(L, \sigma, W)) \tag{4.3}$$

Proof. Just for this proof, we introduce a new model where the access costs, replacing (4.2), are defined by

$$\left(\sum_{l:X_l \in S} w_i \cdot w_l\right) + \binom{w_i}{2}. \tag{4.4}$$

Hence, in this model, we show $\mathit{WOPT}(L, \sigma, W) = \mathit{OPT}(f(L, \sigma, W))$ which is trivially equivalent to (4.3) in the old model.

To see that $\mathit{WOPT}(L, \sigma, W) \geq \mathit{OPT}(f(L, \sigma, W))$, an optimal schedule for (L, σ, W) is transformed to the *OLUP* instance as follows. Remember that a schedule is defined by a list state for every request in the request sequence, denoting the list state in which the request takes place. Let $\sigma_i = X_j$ be the ith request in the *WLUP* instance which is transformed into a sequence of w_j requests in the *OLUP* instance. The list states for all w_j requests will be the same; namely L_i where f is applied on.

What we obtain is a legal schedule for $f(L, \sigma, W)$ with exactly the same cost. This follows from the following observations. If we access X_i in (L, σ, W), this translates to accessing all items $x_{i,1} \ldots x_{i,w_i}$ in $f(L, \sigma, W)$ in turn. In order to access $x_{i,j}$ in our schedule, one has to pass all $x_{l,k}$ with $X_l \in S$ plus all $x_{i,k}$ with $k < j$. Summing up the cost for accessing all items $x_{i,j}$, $j = 1 \ldots w_i$, we obtain (4.4). If two weighted items X_i and X_j are transposed, every item $x_{i,k}$ passes every item $x_{j,l}$ in the *OLUP* schedule. This needs $w_i \cdot w_j$ transpositions.

Proving $\mathit{WOPT}(L, \sigma, W) \leq \mathit{OPT}(f(L, \sigma, W))$ is more involved. Let us start with an optimal schedule for the *OLUP* instance $f(L, \sigma, W)$. We can retransform this instance and its optimal schedule into a *WLUP* instance by treating the items $x_{i,j}$ of $f(L, \sigma, W)$ as weighted items with weight 1. Because there are now weighted items, we write them as $X_{i,j}$.

For a general weight vector W, the total cost of the given schedule depends on the weights and can be expressed as

$$C(W) = \sum_{\substack{i,k,j,l \\ i \leq k}} f_{(i,j),(k,l)} \cdot w_{i,j} w_{k,l} + \sum_{i,j} \binom{w_{i,j}}{2} \cdot |\sigma_{X_i}|. \tag{4.5}$$

We denote by $f_{(i,j),(k,l)}$ the number of times $X_{i,j}$ and $X_{k,l}$ are transposed in the schedule plus the number of times $X_{i,j}$ is in front of $X_{k,l}$ when $X_{k,l}$ is requested and vice versa. The second sum accounts for the second term in (4.4). We can write X_i there instead of $X_{i,j}$ because by construction, the number of requests to $X_{i,j}$ is equal to the number of requests to X_i in the original sequence σ.

If we set all $w_{i,j} = 1$, we have $C(\mathbf{1}) = \mathit{OPT}(f(L, \sigma, W))$. This holds because items with weight one behave exactly like regular items.

Starting from this instance, we will now repeatedly apply a process of merging items by changing their weights. Namely we choose i, j, k, $j \neq k$ such that $w_{i,j} > 0$ and $w_{i,k} > 0$. Let us rewrite (4.5) such that we can more easily detect how its value changes if we change $w_{i,j}$ and $w_{i,k}$.

$$\begin{aligned} C(W) = C_0 \quad &+ \quad C_1 \cdot w_{i,j} + C_2 \cdot w_{i,k} + f_{(i,j),(i,k)} \cdot w_{i,j} w_{i,k} \\ &+ \quad \binom{w_{i,j}}{2} |\sigma_{X_i}| + \binom{w_{i,k}}{2} |\sigma_{X_i}| \end{aligned} \tag{4.6}$$

Here, C_0 denotes all cost independent of both $w_{i,j}$ and $w_{i,k}$. By $C_1 \cdot w_{i,j}$ and $C_2 \cdot w_{i,k}$, we denote cost depending linearly only on one of the two. Assume w.l.o.g. $C_1 \leq C_2$. The process sets the new value of the new $w_{i,j}$ to $w_{i,j} + w_{i,k}$ and sets the new $w_{i,k}$ to zero. To see that the value of $C(W)$ does not increase, observe that C_0 does not change at all and

$$C_1 \cdot (w_{i,j} + w_{i,k}) + C_2 \cdot 0 \leq C_1 \cdot w_{i,j} + C_2 \cdot w_{i,k}, \tag{4.7}$$

and furthermore

$$\begin{aligned} f_{(i,j),(i,k)} \cdot w_{i,j} w_{i,k} + \binom{w_{i,j}}{2} |\sigma_{X_i}| + \binom{w_{i,k}}{2} |\sigma_{X_i}| \geq \\ f_{(i,j),(i,k)} \cdot (w_{i,j} + w_{i,k}) \cdot 0 + \binom{w_{i,j} + w_{i,k}}{2} |\sigma_{X_i}| + \binom{0}{2} |\sigma_{X_i}|. \end{aligned} \tag{4.8}$$

Using $f_{(i,j),(i,k)} \geq |\sigma_{X_i}|$ and

$$\binom{a+b}{2} = \binom{a}{2} + \binom{b}{2} + ab$$

inequality (4.8) is straightforward. This reweighting process must terminate because in each step, the number of items whose weight is set to zero increases by one. What we end up with is an instance where for each i we have exactly one $j \in \{1 \dots w_i\}$ for which $w_{i,j} = w_i$, and all the other $w_{i,k}$, $k \neq j$ are zero. This instance is equivalent to (L, σ, W) we started with. Just rename the $X_{i,j}$ with $w_{i,j} = w_i$ to X_i and forget about the $X_{i,j}$ which have weight zero. Because we did not increase the value of C by changing the weights, we found a schedule for (L, σ, W) whose cost is bounded by $C(\mathbf{1})$, hence we proved $\mathit{WOPT}(L, \sigma, W) \leq \mathit{OPT}(f(L, \sigma, W))$. □

4.3 A Lower Bound

Since no efficient algorithm is known to compute $\mathit{OPT}(L, \sigma)$, one often replaces it by an easily computable lower bound $\overline{\mathit{OPT}}(L, \sigma)$ to show (1.3). In this section, we generalize this idea to weighted items and the *WLUP* problem. We denote the corresponding lower bound for $\mathit{WOPT}(L, \sigma, w)$ by $\overline{\mathit{WOPT}}(L, \sigma, W)$.

Instead of expressing a list state in the usual way, we can write it as the set of relative orderings of all pairs of items. As an example, the list $[X_1 X_3 X_2]$ can be written as $\{X_1 < X_2, X_1 < X_3, X_3 < X_2\}$, where $X_i < X_j$ means that X_i is in front of X_j. The idea of $\overline{\texttt{WOPT}}$ is to drop the condition that the only legal list states are the total orderings of the items. E.g. the state $\{X_1 < X_2, X_2 < X_3, X_3 < X_1\}$ is now legal. Hence, the relative ordering of any pair of items can be chosen independently of any other pair.

The access and update cost are still defined by (4.1) and (4.2). The set S in (4.2) consists of the indices l for which $X_l < X_i$. It is easy to see that this defines a lower bound since a schedule for WOPT is also valid for $\overline{\texttt{WOPT}}$ and has the same cost.

A nice feature of $\overline{\texttt{WOPT}}$ is that it is computable in polynomial time. To see this, remember that the pair $\{X_i, X_j\}$ incurs a cost only if it is swapped or if one of the two items is requested. Since there are

no restrictions on the relative ordering of X_i and X_j in $\overline{\texttt{WOPT}}$, their optimal relative order depends only on the requests to X_i and X_j in the request sequence. Finding the optimal schedule for a pair of items can be done in linear time using one of the algorithms described in the introduction. Because there are $O(n^2)$ pairs of items, the overall running time is $O(n^2 m)$.

Note also that the optimal schedule for an instance on two items does not depend on their weights if both weights are positive. This holds because the total cost of a schedule is of the form $k \cdot (w_i \cdot w_j)$, where $k \in \mathbb{N}$ is independent of w_i and w_j.

Hence, the hardness of *WOPT* seems to stem from the total ordering that must hold at any time in a schedule.

4.4 The Reduction

By Theorem 4.1, it suffices to show a polynomial-time reduction from an $\mathcal{NP}$-hard problem to *WLUP* in order to prove $\mathcal{NP}$-hardness of *OLUP*.

The *minimum feedback arc set problem* (*MINFAS*) [17] will serve well for that purpose. Its decision variant *MINFAS*(G, k) is defined as follows. Given a directed graph $G = (V, E)$ and $k \in \mathbb{N}$, we want to decide whether there is a subset $E' \subseteq E$ with $|E'| \leq k$ such that the graph $G' = (V, E - E')$ is acyclic.

There is a second interpretation which is more natural for our purpose. We interpret an arc pointing from v_i to v_j as a constraint "v_i should be ahead of v_j". What we want to decide is whether there exists a total ordering of the vertices such that less than k of these constraints are unsatisfied.

We show a reduction from *MINFAS*(G, k) to the decision version of *WLUP*, denoted by *WLUP*(L, σ, W, k'). Here we want to decide whether there is a schedule which serves σ from the initial state L at maximal cost k'. More precisely, the reduction consists of a polynomial time computable function f that takes G and k as arguments and returns

must not swap:	$\langle 0 \rangle$	[ab]	$a\eta$
	$\langle 1 \rangle$	[ab]	$baa\eta$
	$\langle 2 \rangle$	[ab]	$(ba)^*$
can swap:	$\langle 3 \rangle$	[ab]	$babb\eta$
	$\langle 4 \rangle$	[ab]	$(ba)^*b$
must swap:	$\langle 5 \rangle$	[ab]	$bbb\eta$

Table 4.1: *optimal behavior on two items*

(L, σ, W) such that

$$\mathit{MINFAS}(G, k) \Leftrightarrow \mathit{WLUP}(L, \sigma, W, \overline{\mathit{WOPT}}(L, \sigma, W) + k). \qquad (4.9)$$

For this section, it is important to understand how an optimal schedule of $\overline{\texttt{WOPT}}$ looks like. As $\overline{\texttt{WOPT}}$ treats all pairs of items independently, we have to investigate how sequences on two items are served optimally. Remember that in the two items case, the behavior does not depend on the weights if they are positive.

We consider a list containing only the items a and b. In order to describe how $\overline{\texttt{WOPT}}$ acts, we must find out in which cases it must, can or must not swap the two items. Table 4.1 gives the answer for a few cases, depending on how the remaining part of the request sequence looks like. We will encounter these cases later in this section and then refer to them by their number in angle brackets, like $\langle 1 \rangle$. The notation is analogous to the one for regular expressions. Thus, $(ba)^*$ denotes the empty sequence or any number of repetitions of ba. The sequence η can be any sequence on a and b. If there is no η at the end of the sequence, we assume that this is the end of the sequence. We say that a (sub)sequence σ is served *perfectly* if we do not break these rules when serving σ.

We now describe the function f which transforms a *MINFAS* instance into a *WLUP* instance. For every vertex v_i of $G = (V, E)$, we have a weighted item V_i with weight $k + 1$. We call them *vertex items* and define $n := |V|$. Additionally, we have two items c and d both with weight one. These are all the items we need.

Let us check briefly that the weights are not too large in order to make Theorem 4.1 work. Clearly, the hard *MINFAS* instances obey $k < |E| \leq |V|^2$. Hence, in those cases, the weights of the items are polynomial in the number of items. Thus, the reduction from *WLUP* to *OLUP* is polynomial.

We set the initial list state to $L = [V_1 V_2 V_3 \dots V_n c\, d]$. The sequence σ is basically of the form

$$(V_1 V_2 V_3 \dots V_n)^*,$$

with additional requests to c and d. It consists of two parts σ' and σ''. The first part is

$$\sigma' := V_1 V_2 V_3 \dots V_n.$$

The second part consists of so-called *arc gadgets*. An arc gadget for $(v_i, v_j) \in E$ basically consists of 6 repetitions of σ' with additional requests to c and d.

Here is the arc gadget for the edge (v_2, v_4) in a graph with five vertices.

$$V_1 c \mathbf{V_2}\, V_3 \mathbf{V_4}\, d\, V_5 V_1 \mathbf{V_2}\, cc\, V_3\, dddccc\, \mathbf{V_4} V_5 (V_1 \mathbf{V_2} V_3 \mathbf{V_4} V_5)^4 \quad (4.10)$$

The following gadget represents the edge (v_4, v_2).

$$V_1 \mathbf{V_2} V_3\, c\, \mathbf{V_4} V_5 V_1 \mathbf{V_2}\, d\, V_3 \mathbf{V_4}\, cc\, V_5 V_1 dddccc\, \mathbf{V_2} V_3 \mathbf{V_4} V_5 (\sigma')^3$$

In general, the first request to c in a gadget for edge (v_i, v_j) is always just in front of the first request to V_i. Hence if $i > j$, the requests to c and d will be placed within the first three repetitions of σ'. The case with five vertices is already general enough. The gadget works in exactly the same way if one replaces V_1, V_3, or V_5 by any number of vertices or omits them.

To finish up the description of the request sequence, let us partition the set of arcs in G into two subsets. E^+ contains the arcs (v_i, v_j) with $i > j$, whereas E^- contains those with $i < j$. In σ'', we have one arc gadget for each arc in G, with the additional restriction that all the arc gadgets of the arcs in E^+ precede those in E^-.

In order to prove (4.9), we first look at some properties of this instance. From now on, we abbreviate the cost $\overline{\mathit{WOPT}}(L, \sigma, W)$ by K. In a schedule that costs no more than $K+k$ units, every pair of items involving a vertex item must be served perfectly. This holds because the cost of such a pair is a multiple of $(k+1)$. Therefore, any non-optimal step involving a vertex item costs at least $k+1$ additional units and there is no way to compensate for that. Consider a pair consisting of two vertex items V_i and V_j, $i < j$. In the initial state, V_i is in front of V_j. Therefore $\overline{\mathit{WOPT}}$, which serves each pair of items independently, has to serve the following request sequence from the initial state $V_i < V_j$:

$$V_i V_j V_i V_j \ldots V_i V_j$$

One way of serving this instance perfectly is to do nothing at all. But there are other perfect schedules for this sequence: In order to stay perfect, we are allowed to swap the two items exactly once (check $\langle 0 \rangle$, $\langle 2 \rangle$, and $\langle 4 \rangle$). Because one should never move V_j in front of V_i when the next request goes to V_i $\langle 0 \rangle$, this swap has to take place before a request to V_j.

It is easy to see that in a schedule which costs at most $K + k$ units, the list state before and after every gadget must have c and d at the end of the list state: Because there are at least three repetitions of σ' at the end of an edge gadget and because of $\langle 5 \rangle$, the items c and d must be at the end of the list. Furthermore, we can assume that all gadgets start with c in front of d. This is certainly true for the first gadget. Moving d in front of c before the first request to c does not make sense. For the other gadgets, we now have a closer look at the requests to c and d only. Note that such a projected gadget ends up with three requests to c and starts with another one to c. Therefore, there is no gain in having d in front of c in between two gadgets. Hence we may assume that any gadget starts in a state having the sublist $[cd]$ at the very end of the list.

To see how *WOPT* serves the gadget for (v_i, v_j), we again have a look at the case (v_2, v_4) in (4.10). Note that we can serve that gadget perfectly if and only if V_2 is in front of V_4 at the first request to d. The crucial point is that in a perfect schedule, when the first request to d takes place, d must still be behind c $\langle 1 \rangle$, while c must be behind V_2 $\langle 1 \rangle$ and d must

be in front of V_4 $\langle 5 \rangle$. Only if V_2 is in front of V_4, we can fulfill these conditions. Note that c and d can pass V_k, $k \notin \{i, j\}$, but they do not have to $\langle 3 \rangle$. At the next request to c, we move c to the front of the list $\langle 3, 5 \rangle$. Later, at the first request of the request triple to d, we move d to the front as well $\langle 5 \rangle$, but c will pass d again later $\langle 5 \rangle$. Because of the additional ${\sigma'}^3$ finishing (4.10), both c and d must be moved behind all vertex items at the end without changing the relative order of c and d $\langle 5 \rangle$.

If V_2 is behind V_4, not all the conditions mentioned in the previous paragraph can be fulfilled at the first request to d. The only way to fix this without paying more than k extra units is to move d in front of c at the first request to d and thus pay one unit more than in the previous case.

Now we are ready to prove (4.9). The easier part is the $\Rightarrow$ direction. If we can get an acyclic graph G' by removing only k arcs, we sort the vertices of G' topologically. The schedule which costs at most $K + k$ looks as follows. We use the initial sequence σ' to rearrange the items V_i according to the topological order $\langle 4 \rangle$. For the rest of σ, we do not change the ordering of the vertex items anymore. Thus, we serve all vertex pairs optimally.

Concerning the arc gadgets, all those corresponding to the arcs in G can be served perfectly. For each arc we removed from G, we have to pay one unit extra. As there are at most k of them, we pay at most $K + k$ units to serve σ.

It remains to prove the $\Leftarrow$ direction of (4.9). There are at most k gadgets which were not served perfectly. We will show that if we remove the arcs corresponding to those gadgets, the resulting graph will be acyclic.

Let C be a subset of E such that C forms a cycle in G. We have to prove that there is at least one arc gadget belonging to C which is not served well. For any arc $e = (v_i, v_j)$ and any list state L, we say e is *open* if we have V_i in front of V_j in L and *closed* otherwise. The arcs in $C \subseteq E^+$ are those which are closed in the initial list. In order to serve such a gadget perfectly, it has to be open when its gadget is served, but remember that we cannot close it anymore afterwards without paying

more that k units extra $\langle 2 \rangle$. The arcs in $C \subseteq E^-$ are open in the initial list. If we want to serve them well, we can not close them before their gadget is served because we cannot reopen them $\langle 2 \rangle$.

Let us have a look at the list just after we served all arc gadgets for E^+ in σ. In order to serve all gadgets belonging to C well, all of them must be open at this time. This means for any arc $e = (v_i, v_j)$ in C that the item V_i must be in front of V_j in the current list. Because C forms a cycle, at least one of them must be closed and hence was not (if it belongs to E^+) or will not be (if it belongs to E^-) served well. This concludes the proof.

Outlook

As the title of this thesis suggests, the list update problem is far from being solved. The gap between 1.50115 and 1.6 is small in absolute values, but the problem remains interesting since closing the gap needs an approach totally different from what has been done to date.

The main goal in the future must be to find good algorithms which are not projective. Actually, the main problem here is to find techniques that allow to analyze such algorithms.

Another direction might be to use complexity theory in order to get more insights. The $\mathcal{NP}$-completeness result discussed in Chapter 4 immediately asks for better approximation algorithms and non-approximability results. A proof that the offline list update problem cannot be approximated within a factor of $1.5 + \varepsilon$ would immediately imply that no polynomial $(1.5 + \varepsilon)$-competitive online algorithm exists unless $\mathcal{P} = \mathcal{NP}$.

At first sight, it seems easy to obtain a c-approximation for *OLUP* for c smaller than 1.5, since 1.5 is a rather trivial lower bound for online algorithms and it is really the online property that boosts the lower bound to 1.5.

On the other hand, online and offline algorithms behave very similar when it comes to analyzing them. In both cases, one knows a lot about the optimal relative order of pairs of items, but it is not clear how to translate this information into total orderings of the items.

Like in the online case, projective algorithms seem to be the only class of algorithms which allow to prove something about them. Extending

projective algorithms to the offline case is straightforward. As a concluding, somewhat speculative remark, this does not seem to lead to algorithms which beat COMB. If this observation can be confirmed in future research, the lower bound of 1.6 for projective algorithms is not caused by the online property at all.

Bibliography

[1] S. Albers (1998), Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing* 27, no. 3, 682–693.

[2] S. Albers (1998), A competitive analysis of the list update problem with lookahead. *Theoretical Computer Science* 197, no. 1–2, 95–109.

[3] S. Albers, and M. Mitzenmacher (1997), Revisiting the COUNTER algorithms for list update. *Information Processing Letters* 64, no. 3, 155–160.

[4] S. Albers and M. Mitzenmacher (1998), Average case analyses of list update algorithms, with applications to data compression. *Algorithmica* 21, 312–329.

[5] S. Albers, B. von Stengel, and R. Werchner (1995), A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters* 56, 135–139.

[6] S. Albers, B. von Stengel, and R. Werchner (1996), List update posets. Manuscript.

[7] S. Albers and J. Westbrook (1998), Self Organizing Data Structures. In A. Fiat, G. J. Woeginger, "Online Algorithms: The State of the Art", *Lecture Notes in Computer Science* 1442, Springer, Berlin, 13–51.

[8] F. d'Amore, A. Marchetti-Spaccamela, and U. Nanni (1993), The weighted list update problem and the lazy adversary. *Theoretical Computer Science* 108, no. 2, 371–384.

[9] C. Ambühl (2000), Offline List Update is $\mathcal{NP}$-hard. *Proceedings of the 8th Annual European Symposium on Algorithms (ESA), Lecture Notes in Computer Science* 1879, 42–51.

[10] C. Ambühl, B. Gärtner, and B. von Stengel (2000), A new lower bound for the list update problem in the partial cost model. *Theoretical Computer Science* 268, no. 1, 3–16.

[11] C. Ambühl, B. Gärtner, and B. von Stengel (2000), Optimal Projective Algorithms for the List Update Problem. *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science* 1853, 305–316.

[12] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson (1994), On the power of randomization in on-line algorithms. *Algorithmica* 11, 2–14.

[13] J. L. Bentley, and C. C. McGeoch (1985), Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM* 28, 404–411.

[14] A. Borodin and R. El-Yaniv (1998), Online Computation and Competitive Analysis. Cambridge University Press, Cambridge.

[15] A. Borodin, N. Linial, and M. E. Saks (1992), An optimal online algorithm for metrical task systems. *Journal of the ACM* 39, 745–763.

[16] F. R. K. Chung, D. J. Hajela, and P. D. Seymour (1988), Self-organizing sequential search and Hilbert's inequality. *Journal of Computer and System Sciences* 36, no. 2, 148–157.

[17] M. R. Garey, D. S. Johnson (1979), Computers and intractability. W. H. Freeman and Co, San Francisco.

[18] G. H. Gonnet, J. I. Munro, and H. Suwanda (1981), Exegesis of self-organizing linear search. *SIAM Journal on Computing* 10, 613–637.

[19] M. Manasse, L. A. McGeoch, and D. Sleator (1988), Competitive algorithms for online problems. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, 322–333.

[20] S. Irani (1991), Two results on the list update problem. *Information Processing Letters* 38, 301–306.

[21] S. Irani (1996), Corrected version of the SPLIT algorithm. *Manuscript.*

[22] A. Karlin, M. Manasse, L. Rudolph, and D. D. Sleator (1988), Competitive snoopy caching. *Algorithmica* 3, 79–119.

[23] R. Karp, and P. Raghavan (1990), unpublished.

[24] D. Koller, N. Megiddo, and B. von Stengel (1994), Fast algorithms for finding randomized strategies in game trees. *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC)*, 750–759.

[25] H. W. Kuhn (1953), Extensive games and the problem of information. In: Contributions to the Theory of Games II, eds. H. W. Kuhn and A. W. Tucker, *Annals of Mathematics Studies* 28, Princeton University Press, Princeton, 193–216.

[26] K. Pietrzak (2001), Ein $O(mn!n^3)$ Algorithmus für das Offline List Update Problem. Semesterarbeit an der ETH Zürich, (1. Teil).

[27] K. Pietrzak (2001), $\mathsf{OPT} = \overline{\mathsf{OPT}}$ ist $\mathcal{NP}$-vollständig. Semesterarbeit an der ETH Zürich, (2. Teil).

[28] N. Reingold, and J. Westbrook (1996), Off-line algorithms for the list update problem. *Information Processing Letters* 60, no. 2, 75–80.

[29] N. Reingold, J. Westbrook, and D. D. Sleator (1994), Randomized competitive algorithms for the list update problem. *Algorithmica* 11, 15–32.

[30] R. Rivest (1976), On self-organizing sequential search heuristics. *Communications of the ACM* 19, 63–67.

[31] D. D. Sleator, and R. E. Tarjan (1985), Amortized efficiency of list update and paging rules. *Communications of the ACM* 28, 202–208.

[32] B. Teia (1993), A lower bound for randomized list update algorithms, *Information Processing Letters* 47, 5–9.

[33] B. von Stengel (1996), Efficient computation of behavior strategies. *Games and Economic Behavior* 14, 220–246.

[34] A. C. Yao (1977), Probabilistic computations: Towards a unified measure of complexity. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, 222–227.

Curriculum Vitae

Christoph Ambühl

born on September 15, 1973 in Willisau-Stadt, Switzerland

1980–1986	**primary school in Willisau-Stadt**
1986–1993	**high school in Willisau and Sursee**
1993–1998	**studies at ETH Zurich** major: Computer Science minor: Operations Research
1998–2002	**PhD student** at Institut for Theoretical Computer Science ETH Zurich