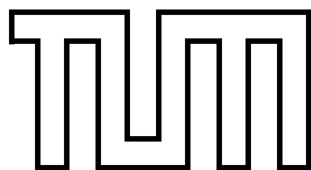


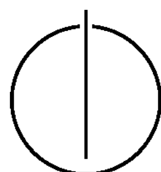
FAKULTÄT FÜR INFORMATIK

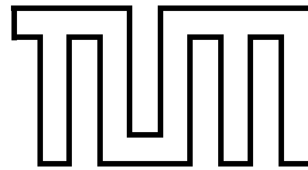
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Verified Analysis of Algorithms for the List Update Problem

Maximilian Paul Louis Haslbeck





FAKULTÄT FÜR INFORMATIK

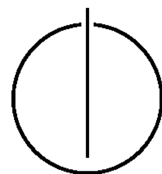
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Verified Analysis of Algorithms for the List Update Problem

Verifizierte Analyse von List Update Algorithmen

Author:	Maximilian Paul Louis Haslbeck
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Prof. Tobias Nipkow, Ph.D.
Submission Date:	September 28th, 2015



Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.
I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, den September 28th, 2015 Maximilian Paul Louis Haslbeck

Acknowledgments

I want to thank Studienstiftung des deutschen Volkes for their financial and ideological sponsorship during my studies. I am very grateful to Prof. Nipkow for introducing me to the world of Isabelle and for supervising my thesis. I warmly thank my parents for their love and support throughout my life.

Also I want to thank Jonas, Sam and Isabelle/HOL for proofreading my thesis.

Abstract

We present a framework for the competitive analysis of online algorithms formalized in Isabelle/HOL and formally verify the analysis of the three most popular algorithms for the List Update Problem: MTF, BIT and TS.

Contents

Acknowledgements **vii**

Abstract **ix**

1 Introduction **1**

2 Competitive Analysis **3**
2.1 Online Algorithms 3
2.2 Competitive Analysis 3
2.3 A Framework for Competitive Analysis in Isabelle/HOL 4
2.3.1 Probability Theory in Isabelle/HOL 4
2.3.2 Competitive Analysis Formalized 6
2.4 The List Update Problem Formalized 8

3 MTF: an Easy Algorithm for the LUP **11**
3.1 Definition of MTF 11
3.2 MTF is 2-competitive 11

4 BIT: an Online Algorithm for the List Update Problem **13**
4.1 Formalization of BIT 13
4.1.1 The Internal State 13
4.1.2 Definition of BIT 14
4.1.3 Properties of BIT's state distribution 14
4.2 BIT is 1.75-competitive 15
4.2.1 Definition of the Locale and Helper Functions 15
4.2.2 The Potential Function 16
4.3 Upper Bound on the Cost of BIT 16
4.4 Main Lemma 17
4.4.1 The Transformation 18
4.4.2 Approximation of the Term for Free exchanges 23
4.4.3 Transformation of the Term for Paid Exchanges 25
4.4.4 Combine the Results 25
4.5 Lift the Result to the Whole Request List 25
4.6 Generalize Competitivness of BIT 26
4.7 Conclusion and Remarks 26

5 List factoring technique **29**
5.1 Another view on the cost of an algorithm for the list update problem 29
5.1.1 The pairwise property 30

5.1.2 Desire for the list factoring technique . . . 31
5.2 List Factoring for OPT . . . 31
5.3 Factoring Lemma . . . 33

6 OPT2: an Optimal Algorithm for Lists of Length 2 **35**
6.1 Formalization of OPT2 . . . 35
6.2 OPT2 is Optimal on Lists of Length 2 . . . 36
6.3 Further properties of OPT2 . . . 38
6.4 Remarks . . . 39

7 TS: another 2-competitve Algorithm **41**
7.1 Definition of TS . . . 41
7.2 Phase Partitioning Technique . . . 43
7.2.1 Regular Expressions Equivalence . . . 43
7.2.2 Analysis of the Phases . . . 44
7.2.3 Phase Partitioning . . . 45

8 Open Questions **47**
8.1 Open to formalize . . . 47
8.2 Open Research Questions . . . 48

9 Conclusion **49**

Bibliography **51**

1 Introduction

The *list update problem* consists of maintaining a singly-linked list of distinct elements and serving access requests. An access has to be served by traversing the list from the front until the requested element has been found. The cost of such an access is thus the position of the requested element. Instantly after the access, it is allowed to bring the requested element nearer to the front of the list by so called *free exchanges*. Any other swap of two consecutive elements in the list costs one unit and is called a *paid exchange*. The goal of an algorithm for the list update problem is to minimize the total cost of serving the request sequence.

The list update problem has received a lot of attention in recent decades and is a fundamental problem in the area of *online algorithms*. An online algorithm has to serve requests from a sequence in order of occurrence without knowledge of future requests. A vast amount of such online problems have been studied, including the paging problem, load balancing and bin packing; stock portfolio selection can also be viewed as an online problem.

For all of these problems a variety of algorithms have been proposed. A natural question is how to measure the quality of these.

A classical answer is *competitive analysis*: Suppose there is an oblivious adversary that knows the future and specifically the complete request sequence. This *offline* adversary is able to compute an optimal strategy. Then algorithms can be compared to this adversary. An algorithm is deemed *competitive* if the ratio between its cost and the optimal cost is bounded by a constant for any request sequence. This bound is called the *competitive ratio* and an algorithm is *c-competitive* if the ratio is at most c. A formal definition follows in Section 2.2.

The list update problem, in particular, has been studied extensively in this framework. The recent survey by Kamali et al. [14] gives an overview of the field.

This thesis is structured as follows.

In Chapter 2 a general framework for competitive analysis of (randomized) online algorithms is specified and its formalization in Isabelle/HOL is presented. We then interpret it for the list update problem.

The rest of this thesis follows the lines of the first two chapters of "Online Computation and Competitive Analysis" by Borodin and El-Yaniv [9] and chronologically presents the most popular algorithms and their verified analysis.

Sleator and Tarjan showed in their seminal paper [23] that the deterministic online algorithm MTF is 2-competitive. In fact this algorithm attains the best possible ratio for deterministic algorithms. This lower bound was established by Irani [13]. The analysis of MTF has already been formalized by Nipkow [16] and was integrated into our framework. In Chapter 3 we formally define MTF and sketch its analysis.

The randomized online algorithm BIT due to Reingold et al. [21] breaks the lower bound

for deterministic algorithms – its formalization and the proof of its competitiveness are presented in Chapter 4.

In Chapter 5 we introduce the list factoring technique, which we use to analyse the algorithm TS in Chapter 7. In order to do that we need some more information about the optimal offline algorithm for lists of length 2. In Chapter 6 we address the algorithm OPT2 which is optimal on such lists and can be stated compactly.

Results for BIT and TS could then be combined to show that the algorithm COMB due to Albers [4] is 1.6-competitive. Ambühl showed that no algorithm could have a better ratio than 1.50115 [6]. Hence there still exists a gap between the hardness result and the known algorithms. Further algorithms and results that may be formalized in future work as well as open research questions are collected in Chapter 8. Chapter 9 finally provides a conclusion to the thesis.

2 Competitive Analysis

For the analysis of online algorithms several techniques have been proposed (Section 2.1); the most prominent being the competitive analysis (Section 2.2) which we formalize in Isabelle/HOL (Section 2.3). We instantiate this framework for the list update problem in Section 2.4 and analyze the most popular algorithms in the remainder of this thesis.

2.1 Online Algorithms

Online algorithms process a request sequence serially and can make their decisions based on the past but without secure information about the future. The study of online algorithms can be seen as a part of "decision making in absence of full information" and is a natural topic of various disciplines such as computer science, economics, finance and decision theory.

Opposed to online algorithms are *offline algorithms* which possess full knowledge of future requests.

There are many examples of online problems studied including the list update problem, caching and load balancing. But the analysis of online algorithms can not only be applied to intrinsically online problems but also for the approximation of many combinatorial optimization problems. Approximation algorithms for NP-hard problems are of special interest.

Early work on the analysis of online algorithms falls into the class of *distributional analysis*: a distribution of events is presumed and the expected cost is studied. This amounts to an average case analysis. Algorithms for the list update problem have already been studied in this way [22, 8, 7].

Research over the past 30 years focused on *competitive analysis*, which we already introduced briefly and will define formally in the next section.

Unfortunately, competitive analysis seems to be too pessimistic compared to the behaviour of the algorithms in practice. The competitive ratios predicted are substantially higher than the observed ones. Recent studies refine the competitive analysis and try to model a phenomenon called *locality of reference*: in any subset of the request sequence the set of requested elements is relatively small. This has already been addressed for the paging problem [10] as well as for the list update problem [3].

2.2 Competitive Analysis

For a request sequence σ let $OPT(\sigma)$ denote the minimal total cost for serving σ, and also let $ALG(\sigma)$ denote the total cost of an online algorithm ALG for serving σ.

Definition 2.1 ([9, sec. 1.1.2])**.** *An online algorithm ALG is* c-competitive *if there is a constant* α *such that for all finite input sequences* σ

$$ALG(\sigma) \leq c \cdot OPT(\sigma) + \alpha.$$

When the additive constant α *is less than or equal to zero, we may say for emphasis that ALG is* strictly c-competitive*.*

The infimum over the set of all values such that ALG is c*-competitive is called the* competitive ratio *of ALG.*

The fact that a c-competitive online algorithm is a c-approximation algorithm, makes results from competitive analysis available to approximation theory.

A possible way of viewing an online problem is that of a game between an online player and an adversary. The adversary creates an malicious request sequence that has to be served by the online player via an online algorithm. There are different graduations of strength of the adversary [9, chapter 4]. We will only consider the oblivious adversary that knows the complete request sequence in advance and for deterministic online algorithms also knows exactly how the online player will react on each requested element. For randomized algorithms the above definition has to be refined.

Definition 2.2 ([9, sec. 2.1])**.** *Let ALG be a randomized online algorithm. Based on the knowledge of ALG – in particular, the probability distribution(s) ALG uses – the oblivious adversary must choose a finite request sequence* σ *in advance. ALG is* c-competitive against an oblivious adversary *if for every such* σ

$$E\left[ALG(\sigma)\right] \leq c \cdot OPT(\sigma) + \alpha.$$

where α *is a constant independent of* σ*, and* $E[\cdot]$ *is the mathematical expectation operator taken with respect to the random choices made by ALG.*

Note if ALG is deterministic this definition collapses to the deterministic one. Similarly as for the deterministic case, the competitive ratio is defined.

2.3 A Framework for Competitive Analysis in Isabelle/HOL

We now present a framework for competitive analysis of randomized online algorithms formalized in Isabelle/HOL. We use the probability theory already present in Isabelle/HOL to state the behaviour of online algorithms and the definition of competitiveness. A short introduction is given in Section 2.3.1. We present the framework in Section 2.3.2, which is parameterized by functions capturing specific behaviour of online problems. Subsequently we give an interpretation of the framework for the list update problem in Section 2.4, by providing these functions.

2.3.1 Probability Theory in Isabelle/HOL

As we want to express statements about distributions of states (e.g. for the randomized online algorithms) and expectations of costs, we need some formalization of such concepts.

Therefor we use the type of *probability mass function (pmf)* formalized in Isabelle/HOL by Hölzl et al. [12, section 4].

They formalize the type pmf on a type α, representing distributions of discrete random variables on α.

The theory provides a function for the support set of the pmf as well as a function *pmf D x* that returns the probability of *x* in the pmf *D*.

Example 2.3. *The theory for instance defines the Bernoulli distribution* $bernoulli_{pmf}$*, a pmf on the type bool which satisfies amongst others the following properties:*
$set_{pmf}\ (bernoulli_{pmf}\ p) \subseteq \{True, False\}$
$pmf\ (bernoulli_{pmf}\ (1\ /\ 2))\ x = 1\ /\ 2$
$[\![0 \leq p; p \leq 1]\!] \Longrightarrow pmf\ (bernoulli_{pmf}\ p)\ True = p$

Furthermore the monadic operators
$bind_{pmf}{::}'a\ pmf \Rightarrow ('a \Rightarrow 'b\ pmf) \Rightarrow 'b\ pmf$,
$return_{pmf}{::}'a \Rightarrow 'a\ pmf$
are provided. $M \ggg f$ is short for *bind-pmf M f*. To apply a function to every element of a probability distribution the function
$map_{pmf}{::}('a \Rightarrow 'b) \Rightarrow 'a\ pmf \Rightarrow 'b\ pmf$
can be used. It is defined in terms of the other two: $map_{pmf}\ f\ M = M \ggg (\lambda x.\ return_{pmf}\ (f\ x))$. With the help of these functions more complex *pmf*s can be synthesized from easier ones.

Example 2.4. *Consider a random experiments that flips two independent different coins, which turn out to show heads (True) with probability* 0.4 *and* 0.5 *respectively. We then say the experiment succeeds if at least one of the coins shows heads. We can model this as follows:*

$$
\begin{aligned}
twocoins \ = \ do \ \{ & \\
& x \leftarrow bernoulli_{pmf}\ (4\ /\ 10); \\
& y \leftarrow bernoulli_{pmf}\ (5\ /\ 10); \\
& return_{pmf}\ (x \lor y) \\
\} &
\end{aligned}
$$

Note that the "do/←"-notation is syntactic sugar for one or more op $\ggg$ *statements. We now can formulate and easily prove*
pmf twocoins True = 7 / 10.

In order to abstract from their development we defined the expectation $E{::}real\ pmf \Rightarrow real$ of a probability distribution over the reals and encapsulated all properties we need about it in our formalization. These properties include monotonicity, congruence and linearity:

Lemma 2.5. *Let X be a probability distribution with finite support set. Then the following statements hold,*
$E[X] = (\sum x{\in}set_{pmf}\ X.\ x * pmf\ X\ x)$
$\forall\, x{\in}set_{pmf}\ X.\ f\ x \leq u\ x \Longrightarrow E[map_{pmf}\ f\ X] \leq E[map_{pmf}\ u\ X]$
$\forall\, x{\in}set_{pmf}\ X.\ f\ x = u\ x \Longrightarrow E[map_{pmf}\ f\ X] = E[map_{pmf}\ u\ X]$
$E[map_{pmf}\ (\lambda x.\ \sum i{\in}A.\ f\ i\ x)\ D] = (\sum i{\in}A.\ E[map_{pmf}\ (f\ i)\ D])$

2.3.2 Competitive Analysis Formalized

In this section we will present our framework for competitive analysis of randomized algorithms.

For capturing the dynamics of an online problem we parameterize our framework the following way:

We assume the problem maintains a *configuration* $c::'configuration$ of some data structure during the service of a request sequence, which in turn is modelled by a list of *'query* elements. An algorithm's reaction to a request is summarized in an *'action*. See the last part of this section for a comparison of our approach with the formal definition of online problem and online algorithm in [9].

In the following it may be beneficial to have the list update problem in mind, but the framework can be instantiated for any other online problem. You may think of the *configuration* being the current state of the list, a *query* being an requested element of the list and an *action* being instructions for paid and free exchanges.

In order to avoid confusion, we denote a state of the list *configuration*, as opposed to the *internal state* of an online algorithm and the *state* tuple consisting of both, a configuration and an internal state.

We require a function $step::'configuration \Rightarrow 'query \Rightarrow 'action \Rightarrow 'configuration$ that returns the configuration after serving a request on a configuration by an action. Function $t::'configuration \Rightarrow 'query \Rightarrow 'action \Rightarrow nat$ determines the costs incurred in that process.

Those two functions suffice to capture the dynamics of a single step of an online problem.

We now can lift these to the total cost T of the serving of a request sequence:

Definition 2.6.

$$\begin{aligned} T\ s\ []\ [] &= 0 \\ T\ s\ (q \cdot qs)\ (a \cdot as) &= t\ s\ q\ a + T\ (step\ s\ q\ a)\ qs\ as \end{aligned}$$

Note that T is only well defined if the length of the request sequence matches the length of the action list.

We model an offline algorithm as a function taking the initial list *init* as well as the whole request sequence *qs* and returning an appropriate list of actions.

Additionally we can define the optimal cost for serving a request sequence *qs* on the initial configuration *init* as the infimum over the costs of all well-formed strategies *as*.

Definition 2.7. $T_{opt}\ init\ qs = Inf\ \{T\ init\ qs\ as \mid |as| = |qs|\}$

In contrast, modelling an online algorithm is a bit more involved, as the algorithms may maintain internal states during their execution. Furthermore, as we need to support randomization we have to express statements about distributions of these states. We first split the definition of an online algorithm into two functions: an online algorithm consists of an initialization phase I and a service phase S.

Firstly, in the initialization phase I the algorithm gets the initial list *init* and may form a probability distribution over internal states *is*. This distribution will later be paired with the list configuration to form a distribution over *state tuples* $(is, init)$.

In the serving phase S the algorithm gets a request q and a specific state tuple (is, s) and returns a probability distribution over the tuples (a, is') consisting of the action taken to serve q and the internal state after serving.

The function *config A qs init n* formalizes the execution of an algorithm by denoting the distribution of state tuples after the nth step of A serving the request sequence qs on the initial list *init*:

Definition 2.8 (config)**.**

$$
\begin{aligned}
config\ (I, _)\ qs\ init\ 0 \quad &= \quad do\ \{ \\
&\qquad is \leftarrow I\ init; \\
&\qquad return_{pmf}\ (is, init) \\
&\quad \} \\
config\ (I, S)\ qs\ init\ (Suc\ n) \quad &= \quad do\ \{ \\
&\qquad (is, s) \leftarrow config\ (I, S)\ qs\ init\ n; \\
&\qquad (a, is') \leftarrow S\ is\ s\ qs_{[n]}; \\
&\qquad return_{pmf}\ (is', step\ s\ qs_{[n]}\ a) \\
&\quad \}
\end{aligned}
$$

For the initial distribution the initial list configuration is paired with the distribution of the internal states the algorithm obtained by I. For the step case: for every state tuple (is, s) in the distribution before the request, execute S together with the requested item obtaining a distribution over the action taken and the following internal state. For any such tuple (a, is') compute the list state s' after serving $qs_{[n]}$ with action a and package it up to form the new state tuple (is', s').

Note that the list configuration s only is manipulated by *step*, thus invariants of step carry over to the second component of any state tuple in the distribution *config*. We can formulate this as an induction rule:

Lemma 2.9. $[\![P\ init;\ \bigwedge s\ q\ a.\ P\ s \Longrightarrow P\ (step\ s\ q\ a)]\!] \Longrightarrow \forall x{\in}set_{pmf}\ (config\ (I, S)\ qs\ init\ n).\ P\ (snd\ x)$

Definition 2.10. *Now we can define the expected cost for one step of the serving by referring to config and mimicking the execution similarly:*

$$
\begin{aligned}
t_{on}\ (I, S)\ qs\ init\ n \quad = \quad & E[do\ \{ \\
&\qquad (is, s) \leftarrow config\ (I, S)\ qs\ init\ n; \\
&\qquad (a, is') \leftarrow S\ is\ s\ qs_{[n]}; \\
&\qquad return_{pmf}\ (t\ s\ qs_{[n]}\ a) \\
&\quad \}]
\end{aligned}
$$

Which can be lifted to the whole request sequence to obtain the total cost:

$T_{on}\ A\ qs\ init \equiv \sum i{<}|qs|.\ t_{on}\ A\ qs\ init\ i$

Finally we can state the competitiveness of an online algorithm formally:

Definition 2.11. $compet\ A\ c\ S0 = (\forall s0{\in}S0.\ \exists b{\geq}0.\ \forall qs.\ T_{on}\ A\ qs\ s0 \leq c * T_{opt}\ s0\ qs + b)$

Please note the order of the quantifiers. For every initial configuration, from a set that can be specified, there is a constant term b such that for every request sequence the inequality

holds. Thus b has to be constant in terms of the request sequence, but may be dependent on the initial configuration.

Comparison to Request-Answer-Games

We relate our formalization of online algorithms and online problems to the Request-Answer-Games in [9, chapter 7].

In our formalization we use as *request set* the whole range of the type *'query* and only have one *answer set* being the range of type *'action*.

Our implicit definition of *randomized online algorithms* via the function *config* is different from the one in [9]. We defined an algorithm step as the transformation of a distribution over state tuples into another distribution over state tuples. In contrast to that Borodin et al. define a randomized online algorithm as a probability distribution over the set of all deterministic online algorithms $\{ALG_x\}$ (with ALG_x meaning the deterministic online algorithm obtained when fixing a stream of random bits x). The author suspects that these notions are equivalent but no proof will be given.

2.4 The List Update Problem Formalized

We interpret the framework for competitive analysis for the list update problem.

For the list update problem we choose the *configuration* to be a list of elements of some type α. An *action* consists of a list of indices p_i::*nat* that specify the paid exchanges and a value *mf* that specifies the free exchange.

For carrying out the list operations we define *swapSuc n xs* to swap consecutive elements at indices n and $n + 1$ in *xs* if in bounds.

Definition 2.12. $swapSuc\ n\ xs = (\textsf{if}\ Suc\ n < |xs|\ \textsf{then}\ xs[n := xs_{[Suc\ n]}, Suc\ n := xs_{[n]}]\ \textsf{else}\ xs)$

This function is lifted to lists of indexes (forming *swapSucs*), with the convention that the list is processed from the back, i.e. the index located at the last position of the list is swapped first.

The free exchange *mtf2 n q c* of the requested element *q n* positions to the front in *c* is defined by bubbling *q* to the front via *swapSucs*. This enables us to use the lemmas proven about *swapSucs* also for *mtf2*. Note that *index xs x* is the index of the first occurrence of *x* in *xs*.

Definition 2.13. $mtf2\ n\ x\ c = (\textsf{if}\ x \in set\ c\ \textsf{then}\ swapSucs\ [index\ c\ x - n..{<}index\ c\ x]\ c\ \textsf{else}\ c)$

Finally we can state *step* and *t*:

Definition 2.14.
$step\ s\ q\ a = (\textsf{let}\ (k, sws) = a\ \textsf{in}\ mtf2\ k\ q\ (swapSucs\ sws\ s))$
$t\ s\ q\ a = (\textsf{let}\ (mf, sws) = a\ \textsf{in}\ index\ (swapSucs\ sws\ s)\ q + 1 + |sws|)$
$t^*\ s\ q\ a = (\textsf{let}\ (mf, sws) = a\ \textsf{in}\ index\ (swapSucs\ sws\ s)\ q + |sws|)$

step first executes the paid exchanges and then the free exchange. The total cost of a step measured by t is the position of the requested element plus 1 and the number of paid exchanges executed. This holds for the full cost model, which will be used in chapter 3 and 4.

For the partial cost model, which will be used from Chapter 5 on, we also define t^* to be the cost for a single step: The only change is that the cost of the request omits the additional cost of 1.

Note that we interpret the framework for *step* and t obtaining the functions T, T_{opt} etc, when we interpret it for the partial cost model (i.e. for *step* and t) we obtain functions T^*, T^*_{opt} etc.

Concerning the second component of the distribution of the state tuples of any algorithm (the lists maintained by the algorithm), we can use the invariant preservation. As we know that step only permutes the initial list configuration, we can easily show the following lemma by the induction principle 2.9:

Lemma 2.15.

$\forall\, x \in set_{pmf}\ (config\ (I, S)\ \sigma\ init\ n).$

$\ \ set\ (snd\ x) = set\ init\ \wedge$

$\ \ distinct\ (snd\ x) = distinct\ init \wedge |snd\ x| = |init|$

Now we have the tools in place to analyse several algorithms for the list update problem: we analyze the deterministic online algorithm Move to Front (chapter 3), the randomized online algorithm BIT (chapter 4), the offline algorithm OPT2 (chapter 6), and finally the deterministic online algorithm TS (chapter 7).

3 MTF: an Easy Algorithm for the LUP

3.1 Definition of MTF

The first algorithm we consider is MTF (Move To Front). As its name suggests this algorithm moves every requested element to the front by free exchanges.

Definition 3.1 (MTF informal)**.** *After accessing an element, move it to the front of the list, without changing the relative order of the other items.*

The formal definition of MTF that fits into our framework is straightforward. The internal state is empty and the action taken to serve a request is constant.

Definition 3.2 (MTF)**.**

$$\begin{array}{lcl} \textit{MTF-init } s & = & \textit{return}_{pmf}\ () \\ \textit{MTF-step is } s\ q & = & \textit{return}_{pmf}\ ((|s|, []), ()) \\ \textit{MTF} & = & (\textit{MTF-init}, \textit{MTF-step}) \end{array}$$

3.2 MTF is 2-competitive

As the proof of this section has already been formalized [16] and was only integrated into the framework, we will not go into great detail about it. Nevertheless, we present the following two concepts and the structure of the proof as we will reuse them for the analysis of BIT in Chapter 4.

Amortized Analysis

We use the amortized complexity analysis with a potential function which has been formalized in Isabelle/HOL [17]. Essentially we use the following lemma from it:

Lemma 3.3.

$$[\![\Phi\ 0 = 0;\ \bigwedge n.\ 0 \leq \Phi\ n;\ \bigwedge n.\ t\ n + \Phi\ (n+1) - \Phi\ n \leq u\ n]\!] \Longrightarrow (\sum i{<}n.\ t\ i) \leq (\sum i{<}n.\ u\ i)$$

It states that, given an appropriate potential function Φ and showing that the amortized cost can be bounded by u, we can conclude that u is indeed an upper bound for the real cost t.

Inversions

A key concept we exploit in the following proof is the way how to compare the lists maintained by two algorithms. We introduce the concept of inversions:

Definition 3.4. *Suppose xs is a permutation of the list ys and $x < y$ in xs denotes that element x precedes y in xs, then we define the* inversions *between xs and ys as,*
$Inv\ xs\ ys = \{(x, y) \mid x < y \text{ in } xs \wedge y < x \text{ in } ys\}$

A natural interpretation of the number of inversions of two lists is the number of swaps needed to transform one list into the other. Thus it also bounds the difference of the indexes of an element in both lists. This fact will later be exploited

The proof

The analysis of MTF was first conducted by Sleator and Tarjan [23].

The proof in [16] begins with fixing a request sequence *qs* and an adversary strategy *as* of an adversary *A*. A bunch of helper functions are defined including c_A (the cost incurred by the access of the requested element for *A*), p_A (the cost for paid exchanges of *A*) and f_A (the number of positions A moves the requested item to the front), $s_A n$ the list configuration of *A* after step *n*, $t_A\ n$ the cost of *A* in step *n* and $s_{mtf}\ n$ the list configuration of *MTF* after step *n*, . Then an amortized argument is used for showing that MTF's total cost can be bounded by the double of the cost incurred by *as* serving *qs*.

For that purpose a potential function is defined that maps the configurations of A and MTF to the number of inversions between the two lists.
$\Phi\ n = |Inv\ (s_A\ n)\ (s_{mtf}\ n)|$
The demanding part of the proof – showing that the amortized cost of a single step of MTF can be bounded by the cost of A – will be ommited here. The proof obligation is:

Lemma 3.5. $t_{mtf}\ n + \Phi\ (n + 1) - \Phi\ n \leq 2 * c_A\ n - 1 + p_A\ n - f_A\ n$

Together with the potential function technique (Lemma 3.3) it can be used to show the 2-competitiveness of MTF against any adversary A and thus against the optimal offline algorithm:

Theorem 3.6.
$[\![init \neq [];\ distinct\ init;\ set\ qs \subseteq set\ init]\!] \Longrightarrow T_{on}\ MTF\ qs\ init \leq (2 - 1\ /\ |init|) * T_{opt}\ init\ qs$

and thus

Theorem 3.7. *compet MTF 2* $\{init \mid distinct\ init\}$

This concludes this section. It is notable that MTF, albeit its simplicity, attains the best competitive ratio possible for deterministic online algorithms [13].

4 BIT: an Online Algorithm for the List Update Problem

In this chapter we study a simple randomized algorithm for the list update problem called BIT due to Reingold and Westbrook [19]:

Definition 4.1 (BIT informal)**.** *BIT keeps for every element x on the list a mod2-counter,* $b_{[x]}$*: For each element x on the list, randomly initialize its bit* $b_{[x]}$ *independently and uniformly. When a request to access an element x is given, first complement its bit* $b_{[x]}$*. Then, if* $b_{[x]} =$ *True, move x to the front; otherwise (*$b_{[x]} =$ *False) do nothing.*

It turns out that this algorithm breaks the 2-competitive barrier mentioned in the introduction and it can be shown that BIT is 1.75-competitive.

In this chapter we first provide the necessary machinery to define BIT, then give a definition of BIT that fits into our framework and finally prove that BIT is 1.75-competitive.

Therefor we proceed as in the proof of Theorem 3.7: an amortized analysis is employed with a potential function involving inversions. Nevertheless, as the proof talks about expectations rather than concrete values it is a bit more intricate.

4.1 Formalization of BIT

In contrast to MTF, BIT maintains an internal state.

In this section we first define a uniform probability distribution over bit vectors of length n, and call it *L n*. We will prove some basic properties about it and will use it in order to define BIT. As BIT flips some bit during a request, such a function has to be provided and the effect on the probability distribution has to be examined.

4.1.1 The Internal State

We define *L n* as the uniform distribution over bit vectors (bool list) of length *n* recursively and prove the following characteristic lemmas about it:

Definition 4.2.

$L\ 0 = return_{pmf}\ []$

$L\ (Suc\ n) = do\ \{$

$\quad xs \leftarrow L\ n;$

$\quad x \leftarrow bernoulli_{pmf}\ (5\ /\ 10);$

$\quad return_{pmf}\ (x \cdot xs)$

$\}$

Lemma 4.3.
$finite\ (set_{pmf}\ (L\ n))$
$set_{pmf}\ (L\ n) = \{x \mid |x| = n\}$
$n < l \Longrightarrow map_{pmf}\ (\lambda y.\ y_{[n]})\ (L\ l) = bernoulli_{pmf}\ (5\ /\ 10)$

As expected, if we look one specific bit, in the distribution of all bit vectors of length n, it has equal probability to be *True* and *False*.

We use the notation $[\![b]\!](n)$ for the value of the nth bit in the bit vector b as a natural number.

We further define the simple function *flip i b* that flips the ith bit of the bit vector b.

We then see that if we apply *flip i* on every member of the probability distribution $L\ n$ we obtain again the same probability distribution:

Lemma 4.4. $map_{pmf}\ (flip\ i)\ (L\ n) = L\ n$

4.1.2 Definition of BIT

With $L\ n$ and *flip* in place we are able to define the BIT algorithm in our framework with an initialization and a step function:

Definition 4.5 (BIT). $BIT\text{-}init\ init = L\ |init|$
$BIT\text{-}step\ is\ s\ q = (\mathbf{let}\ a = (\mathbf{if}\ is_{[q]}\ \mathbf{then}\ 0\ \mathbf{else}\ |s|, [])\ \mathbf{in}\ return_{pmf}\ (a, flip\ q\ is))$
$BIT \equiv (BIT\text{-}init, BIT\text{-}step)$

To initialize BIT, it generates the probability distribution $L\ n$ over the possible bit vectors.

The step function of BIT formalizes what has been stated before informally: given the current internal state *is*, list configuration s and the requested element q: return a probability distribution of the appropriate action taken by BIT and the resulting internal state. As we have mentioned BIT is a barely random algorithm, thus the step function deterministically yields one possible reaction. Note that "behavioural algorithms" (like RMTF) might use randomization here.

Together, the initialization and step function form the online algorithm *BIT*.

4.1.3 Properties of BIT's state distribution

We can prove some interesting properties about BIT:

Note that BIT does not use paid exchanges: $\forall\ ((f, p), is')\in set_{pmf}\ (BIT\text{-}step\ is\ s\ q).\ p = []$.

Throughout the execution of BIT the distribution of internal states stays the same:

Lemma 4.6. $map_{pmf}\ fst\ (config\ (BIT\text{-}init, BIT\text{-}step)\ qs\ init\ n) = L\ |init|$.

This stems from the fact that all internal states get the requested element's bit flipped and as we have seen this does not alter the distribution. This property can be established by an induction on n and lemma 4.4.

It follows directly that the internal state maintains the same length during the serving of the requests.

$\forall x \in set_{pmf}$ (*config* (*BIT-init*, *BIT-step*) *qs init n*). $|fst\ x| = |init|$

Concerning the second component of the distribution of the state tuples of BIT (the lists maintained by BIT) we already know Lemma 2.15, i.e. that it is always a permutation of the initial list configuration.

4.2 BIT is 1.75-competitive

Now that we defined BIT, we can tackle the proof of its competitiveness.

The proof works similar as the one for MTF: first we fix an adversary A and the request sequence, then we show that BIT's expected cost can be bounded by $\frac{7}{4}$ times the cost of A. As we can prove this for any adversary, it also holds against the optimal offline strategy and we can conclude that BIT is $\frac{7}{4}$-competitive.

The strategy is clear, but it still is a long way to go:

4.2.1 Definition of the Locale and Helper Functions

As mentioned we fix a request sequence *qs* and an adversary $\bar{A}$'s actions *acts*, appropriately chosen (it has to hold $|acts| = |qs|$). Furthermore we restrict the initial list to be a permutation of the numbers $\{0..<|init|\}$. This seems to be a natural restriction, as any list of items could be transformed into such a permutation by a bijection between the elements of the list and their position in the list.

This time we want to exclude paid exchanges of $\bar{A}$ which are out of bounds. It is clear that these do not have an effect on $\bar{A}$'s list (c.f. definition 2.12) but only add to $\bar{A}$'s cost. To that end by filtering out these paid exchanges one obtains an algorithm A. We formally verify that the costs won't rise while the effect on the list stays the same:

Similar to the proof of MTF we now define a whole bunch of helper functions.

First we define the functions $free_A$, f_A *n* being the list of free exchange actions and the number of positions the requested element is moved forward in the *n*th step. Furthermore $paid_A$ ($paid_{\bar{A}}$), s_A *n* ($s_{\bar{A}}$ *n*), c_A ($c_{\bar{A}}$), p_A ($p_{\bar{A}}$), t_A ($t_{\bar{A}}$), being the list of paid exchanges, the list after the *n*th step, cost of the free exchanges in *n*th step, number of paid exchanges in the *n*th step, and total cost in the *n*th step of A and $\bar{A}$ respectively.

Then we verify that the filtering does not rise the cost while the effect stays the same:

Theorem 4.7.

$n \le |qs| \Longrightarrow T_A\ n \le T_{\bar{A}}\ n$

$n < |qs| \Longrightarrow s_A\ n = s_{\bar{A}}\ n$

In the following we show that BIT is competitive to A (the filtered strategy), which then implies that it is also competitive against the strategy $\bar{A}$.

Furthermore we define a function that determines the list configuration at the point in time just after the *m*th paid exchange has been executed while the *n*th element is requested.

In the proof it will be important which element is swapped in each step. This is determined by the function *gebub* and clearly these elements are always in bounds:
$[\![n < |paid_A|; m < |paid_{A[n]}|]\!] \Longrightarrow gebub\ n\ m < |init|$.

4.2.2 The Potential Function

As already mentioned we will prove the result via an argument using the potential function technique. In contrast to the proof of MTF the potential function will be a bit more complicated and takes into account the internal state maintained by BIT.

Similar to the potential function in the proof of Theorem 3.5, the potential function we use here counts the number of inversions between BIT and A's lists. Recall, an inversion is an ordered pair of items (x, y) such that x precedes y in BIT's list and y precedes x in A's list. We define w(x,y), the weight of the inversion (x, y), as the number of accesses to y before y passes x in BIT's list. As mentioned, $[\![b]\!](y)$ denotes the value of the bit of element y as a natural number. Since y passes x by moving to the front, by the definition of BIT and $[\![b]\!](y)$, we have $w\ (x, y) = [\![b]\!](y)+1$.

Finally we define the potential for step n with a pair of internal state and list configuration:

Definition 4.8. $\varphi\ n\ (b, c) = (\sum\ (x, y) \in Inv\ c\ (s_A\ n).\ [\![b]\!](y)+1)$

When now define the potential as the expectation of φ over the probability distribution of BIT's states after the nth request:

Definition 4.9. $\Phi\ n = E[map_{pmf}\ (\varphi\ n)\ (config\ BIT\ qs\ init\ n)]$

Φ is clearly nonnegative ($0 \leq \Phi\ n$), and as BIT and A start with the same initial list the initial potential $\Phi\ 0 = 0$.

4.3 Upper Bound on the Cost of BIT

The worst case cost for a single step of BIT occurs when the requested element is located at the end of the list. Thus BIT's cost can be bounded by the list length, this simple result can be lifted to bound BIT's total cost from above:

Theorem 4.10. $\forall\, i<n.\ qs_{[i]} \in set\ init \Longrightarrow T_{BIT}\ n \leq n * |init|$

4.4 Main Lemma

So now we are ready to state the main lemma in this chapter and formalize its proof.

Theorem 4.11. $t_{BIT}\ n + \Phi\ (n + 1) - \Phi\ n \leq 7\ /\ 4 * t_A\ n - 3\ /\ 4$

Proof. We first consider the degenerate case that the initial list is empty, then the potential function is always zero, cost of BIT is 1 and cost of A is a least one.

In every step of the algorithm both BIT and the algorithm A serve the same request. This involves the movement to front of BIT as well as the paid and free exchanges by A.
To name the respective list configurations and internal states we have the following conventions, which are depicted in Figure 4.1: A's list at the beginning of the request is called xs, after the paid exchanges xs' and after the free exchanges xs''. The position of the requested element after the paid exchanges (which also is part of A's cost) is denoted by k, whereas the position it is moved to by free exchanges by k'.
As BIT does not use paid exchanges we name its list at the beginning of the request ys and after the potential move to front ys'. The position of the requested element is before the request is labelled l. The internal states of BIT are called b and b' before and respectively after the processing of the step. Note that in any case the requested element's bit is flipped.

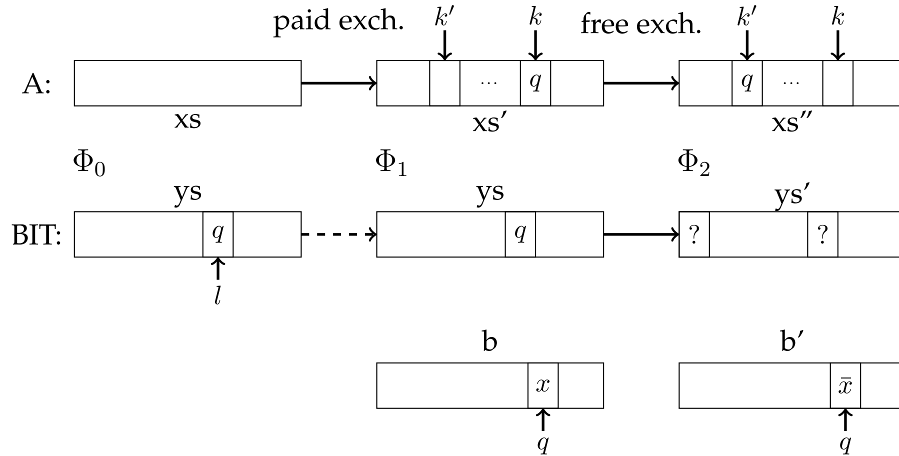


Figure 4.1: Overview of the conventions of naming.

We denote the probability distribution over the state tuples of BIT by D:
$D \equiv config\ (BIT\text{-}init, BIT\text{-}step)\ qs\ init\ n.$
The current goal now is to show that the cost of BIT plus the change in potential is bounded by $7\ /\ 4$ the cost of A.
As a first step, we transform the lhs of our goal to obtain an certain expectation over the distribution of BIT's state:

$$t_{BIT}\ n + \Phi\ (n + 1) - \Phi\ n = E[map_{pmf}\ (\lambda x.\ cost\ x + \Phi_2\ x - \Phi_0\ x)\ D] \tag{4.1}$$

This is obtained by unfolding the definitions (of Φ and t_{BIT}), and linearity lemma of E. For now, to calculate the costs of BIT, we need to know exactly the list configuration of BIT. In the next proof step we will approximate this cost for every pair of internal state and list configuration without using information about the list configuration. This approximation then will only depend on the BIT's internal state (the bit vector) and information from A's strategy (position of the requested element, etc.).

As a preview of the total proof's structure: this approximation can then be used to take the expectation of the cost over the distribution of the first component of BIT's states, which we already know: we have shown in Lemma 4.6 that this distribution is always $L\ n$.

Thus we can take the expectation over all possible bit vectors and obtain the desired result by bounding the resulting term.

4.4.1 The Transformation

But let us now turn to the approximation of BIT's cost independent of its list configuration: What we will show in the following is this lemma:

Lemma 4.12.
$\forall x \in set_{pmf}\ D.$
$\quad cost\ x + \Phi_2\ x - \Phi_0\ x$
$\quad \leq k + 1 +$
$\quad\quad (\textbf{if}\ q \in set\ init$
$\quad\quad\ \ \textbf{then if}\ (fst\ x)_{[q]}\ \textbf{then}\ k - k'\ \textbf{else}\ \sum j{<}k'.\ [\![fst\ x]\!](xs'_{[j]}){+}1\ \textbf{else}\ 0) +$
$\quad\quad (\sum i{<}|paid_{A[n]}|.\ [\![fst\ x]\!](gebub\ n\ i){+}1)$

Proof. For any state x that is a possible state in the distribution D we show that we can approximate the cost of the nth step plus the change in potential by $k + 1$ (which is the cost of A for the list access), a term for the paid exchanges A processes and a term for the free exchanges both A and BIT execute.

Note that $\Phi_2\ x$ measures the inversions after the request (i.e. in xs'' and ys'), whereas $\Phi_1\ x$ counts the inversions after the paid exchanges (i.e. in xs' and ys) and $\Phi_0\ x$ counts the inversions before the request (i.e. in xs and ys).

First we examine how inversions evolve during the paid exchange phase of A.

Upper bound of the inversions created by paid exchanges of A

Consider what happens during the paid exchange phase of A. By definition every paid exchange of A costs one unit, does not affect BIT's state and only swaps two elements of A's list. Thus with every paid exchange maximum one new inversion can be created. This inversion will then be weighted according to whether the second component's bit is set in BIT's internal state.

Essentially, the potential after the paid exchanges can thus be bounded from above by the potential before the paid exchange plus the number of paid exchanges weighted according to the swapped element's bit:

$\Phi_1\ x \leq \Phi_0\ x + (\sum i{<}|paid_{A[n]}|.\ [\![b]\!](gebub\ n\ i){+}1)$

This theorem can easily be established by an induction on the length of the paid exchange list. The already prepared function *s'-A n i* gives the state in the nth request after the ith paid exchange. *gebub n i* specifies the element that was swapped in the *n*th request at the *i*th paid exchange, and thus determines (in the case an inversion was created by that swap) the weight of that inversion.

Hence we have expressed the potential after the paid exchanges, and thus before the free exchanges, by the initial potential and a sum over some bits of BIT's internal state.

Next we investigate on how the cost of BIT can be approximated.

Upper bound for the costs of BIT

We worked off the paid exchanges of A. As every paid exchange costs A one unit, the change in potential is compensated so far.

Now, following our conventions, we denote the position of the requested element q in A's list by k; that is, A's access cost is $k + 1$. Let I count the number of inversions of the form (x, q) of either weight. As mentioned in section 3.2 we see that q is located in BIT's list at most at location $k + I$.

Let *cost x* denote the cost of BIT when operating on the state tuple x, then we can bound this cost following the idea from above: $\mathit{cost}\ x \leq k + 1 + I$.

Upper bound for inversions generated by free exchanges

We are in a good position as we can determine exactly the effects of A's transportations and for BIT there are only two possibilities (move q to front or do nothing) which just depends on BIT's internal state.

In the case that q is not in the list, neither BIT's internal state, nor BIT's and A's lists change, thus the potential does not change.

In the case that q is in the list, we want to express the change of potential independently of BIT's current list configuration.

To analyze the change, we express the change as $\Phi_2\ x - \Phi_1\ x = A - B + C$, where A is the contribution of new inversions created, B is the contribution of old inversions removed and C is the contribution of old inversions that remain but change their weight. Formally

$A \equiv \sum (x, y){\in}\mathit{Inv}\ ys'\ xs'' - \mathit{Inv}\ ys\ xs'.\ [\![b']\!](y){+}1$

$B \equiv \sum (x, y){\in}\mathit{Inv}\ ys\ xs' - \mathit{Inv}\ ys'\ xs''.\ [\![b]\!](y){+}1$

$C \equiv \sum (x, y){\in}\mathit{Inv}\ ys'\ xs'' \cap \mathit{Inv}\ ys\ xs'.\ [\![b']\!](y){+}1 - [\![b]\!](y){+}1$

We first consider B and C, thus the inversions that get removed or change their weight. We have to examine two cases:

Either the requested elements bit is set ($b_{[q]} = \mathit{True}$): then q stays in place in BIT's list; however, since $b_{[q]}$ is flipped to *False*, each inversion of the form (x, q) changes its weight from 2 to 1. Since q stays in place, BIT does not eliminate any old inversion. Furthermore, A's free

transportations will not eliminate any old inversions that are counted by I. Consequently $C = -I$. Only inversions of the form (q, x) may be touched by A, hence $0 \leq B$.

We present this first case, in which the bit is set and thus BIT leaves the list unchanged, more formally: $ys' = ys$. It is obvious that $0 \leq B$, as B counts inversions. Furthermore the contribution of inversions that change their weight can be determined this way:

$$
\begin{aligned}
& C \\
= \; & \textstyle\sum (x, y) \in Inv\ ys'\ xs'' \cap Inv\ ys\ xs'.\ [\![b']\!](y)+1 - [\![b]\!](y)+1 \\
= \; & \textstyle\sum (x, y) \in Inv\ ys'\ xs'' \cap Inv\ ys'\ xs'.\ [\![b']\!](y)+1 - [\![b]\!](y)+1 \\
= \; & \textstyle\sum (x, y) \in Inv\ ys'\ xs'' \cap Inv\ ys'\ xs'.\ \textsf{if}\ q = y\ \textsf{then} -1\ \textsf{else}\ 0 \\
= \; & \textstyle\sum (x, y) \in \{(x, y) \mid (x, y) \in Inv\ ys'\ xs'' \cap Inv\ ys'\ xs' \wedge y = q\} \cup \\
& \qquad \{(x, y) \mid (x, y) \in Inv\ ys'\ xs'' \cap Inv\ ys'\ xs' \wedge \\
& \qquad\qquad y \neq q\}.\ \textsf{if}\ q = y\ \textsf{then} -1\ \textsf{else}\ 0 \\
= \; & (\textstyle\sum (x, y) \in \{(x, y) \mid (x, y) \in Inv\ ys'\ xs'' \cap Inv\ ys'\ xs' \wedge y = q\}. -1) + \\
& (\textstyle\sum (x, y) \in \{(x, y) \mid (x, y) \in Inv\ ys'\ xs'' \cap Inv\ ys'\ xs' \wedge y \neq q\}.\ 0) \\
= \; & -|\{(x, y) \mid (x, y) \in Inv\ ys'\ xs'' \cap Inv\ ys'\ xs' \wedge y = q\}| \\
= \; & -I
\end{aligned}
$$

First we unfold the definition of C, being the contribution of inversions that exist both before and after the free exchanges. As BIT's list does not change, ys can be replaced by ys'. Then one can observe that the difference in the sum is nonzero iff the second component of the inversion is the requested element: in that case the weight decreases by 1. Now we split the set on whether it has the requested element as a second component; then we see that the second part has no contribution and the first one matches the set of inversions of the form (z, q) before the free exchange occurred (being exactly what is counted by I). This is the case because algorithm A only moves q further to the front, which cannot eliminate any inversion of that form.

If ($b_{[q]} = False$), q is moved to front in BIT's list. This eliminates all inversions of the form (x, q). These were of weight 1 prior to the access and counted by I. There might be more inversions eliminated by A's transportation but we have at least: $I \leq B$. Clearly there are no inversions that change their weight: $C = 0$.

We do not present this case formally as it is proven similarly to the first case. The interested reader may look at the theory files for more details.

In both cases we learn:

$C - B \leq -I$

As we saw, the approximation of the cost of BIT involves a positive I, in the final summation it will cancel out again.

Now we have to find an approximation for A, the newly created inversions.

Remark 4.13. *Consider we move the element c further to the front in a list. This move leaves the relative order of all other elements untouched and thus does not create nor modify an inversion between two other elements a and b. Thus, when we look at free exchanges only inversions involving the requested element q could be created: either of the form* (q, z) *or* (z, q)*.*

In case $b_{[q]} = False$, q is moved to front.

As we learnt that in this case q is moved to front no inversion of the form (z, q) will be generated. Thus we can investigate on which inversions of the form (q, z) are newly generated.

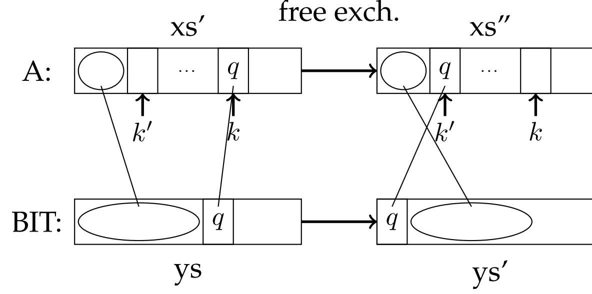


Figure 4.2: Situation when BIT moves q to the front

Figure 4.2 depicts the situation: Any element z that is located in front of position k' in A's list was not in an inversion of the form (q, z) with q before the free exchanges, but is afterwards.

Note that this is only true if z was also in front of q in BIT's list. Consequently we overapproximate the set of newly created inversions:

Lemma 4.14. *Inv ys′ xs″ − Inv ys xs′* $\subseteq \{(a, b) \mid a = q \wedge b \neq q \wedge \mathit{index}\ xs'\ b < k'\}$

To determine the contribution of these newly generated inversion to A we sum up over all positions $\{0..<k\}$ the bit of the respective element.

$$
\begin{aligned}
& A \\
= \; & \textstyle\sum (x, y) \in \mathit{Inv}\ ys'\ xs'' - \mathit{Inv}\ ys\ xs'.\ [\![b']\!](y)+1 \\
\leq \; & \textstyle\sum (x, y) \in \{(a, b) \mid a = q \wedge b \neq q \wedge \mathit{index}\ xs'\ b < k'\}.\ [\![b']\!](y)+1 \\
= \; & \textstyle\sum (x, y) \in \{(a, b) \mid a = q \wedge b \neq q \wedge \mathit{index}\ xs'\ b < k'\}.\ [\![b]\!](y)+1 \\
\leq \; & \textstyle\sum (x, y) \in \{(q, b) \mid \mathit{index}\ xs'\ b < k'\}.\ [\![b]\!](y)+1 \\
= \; & \textstyle\sum j<k'.\ [\![b]\!](xs'_{[j]})+1
\end{aligned}
$$

First the definition of A is unfolded and we use the overapproximation of the set of new inversions (Lemma 4.14). Afterwards we use the knowledge that the second component is not the requested element to infer that the bit vector is not flipped in the requested positions; this condition is dropped afterwards. Finally we rearrange the sum in order to obtain the desired form.

A similar argument works in the case that BIT does not move q to the front, i.e. $b_{[q]} = \mathit{True}$:

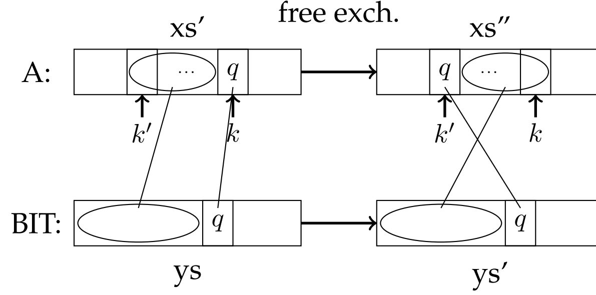


Figure 4.3: Situation when BIT keeps q at its place

In this case Figure 4.3 depicts the situation: now only A executes a free exchange while BIT's list stays untouched. Thus only elements that are located between position k' and k will generate new inversions of the form (z, q).
Again, as not all elements between positions k' and k in A's list might precede q in BIT's list, this is an overapproximation.

Lemma 4.15. $\mathit{Inv}\ ys'\ xs'' - \mathit{Inv}\ ys\ xs' \subseteq \{(xs'_{[i]}, q) \mid i \in \{k'..<k\}\}$

$$
\begin{array}{ll}
 & A \\
= & \sum (x, y) \in \mathit{Inv}\ ys'\ xs'' - \mathit{Inv}\ ys\ xs'.\ [\![b']\!](y)+1 \\
\leq & \sum (z, y) \in \{(xs'_{[i]}, q) \mid i \in \{k'..<k\}\}.\ [\![\mathit{flip}\ q\ b]\!](y)+1 \\
= & \sum z \in \{(xs'_{[i]}, q) \mid i \in \{k'..<k\}\}.\ [\![\mathit{flip}\ q\ b]\!](q)+1 \\
= & \sum y \in \{(xs'_{[i]}, q) \mid i \in \{k'..<k\}\}.\ 1 \\
= & k - k'
\end{array}
$$

We first unfold the definition of A then apply the overapproximation 4.15. The weight of the newly created inversions are determined by the second component of the pair. As this is the requested element, and thus the bit unset, the weight is always 1. Finally we obtain $A \leq k - k'$ for this case.

Merging the findings about A, B and C, we obtain the desired result.
$\Phi_2\ x - \Phi_1\ x \leq -I + (\textbf{if}\ q \in \mathit{set}\ \mathit{init}\ \textbf{then if}\ b_{[q]}\ \textbf{then}\ k - k'\ \textbf{else} \sum j{<}k'.\ [\![b]\!](xs'_{[j]})+1\ \textbf{else}\ 0)$

We can put together the results about the paid exchanges and the free exchanges:

$$
\begin{array}{l}
\forall x \in \mathit{set}_{pmf}\ D. \\
\quad \mathit{cost}\ x + \Phi_2\ x - \Phi_0\ x \\
\quad \leq k + 1 + \\
\qquad (\textbf{if}\ q \in \mathit{set}\ \mathit{init} \\
\qquad\ \textbf{then if}\ (\mathit{fst}\ x)_{[q]}\ \textbf{then}\ k - k'\ \textbf{else} \sum j{<}k'.\ [\![\mathit{fst}\ x]\!](xs'_{[j]})+1\ \textbf{else}\ 0) + \\
\qquad (\sum i{<}|\mathit{paid}_{A[n]}|.\ [\![\mathit{fst}\ x]\!](\mathit{gebub}\ n\ i)+1)
\end{array}
$$

□

We have seen how to bound the cost of BIT and the change in potential for every possible state of BIT independently of the actual list configuration of BIT.

In a next step, in order to obtain a concrete bound for the total cost of BIT, we have to take the expectation of every term over the probability distribution of the state tuples of BIT,.

4.4.2 Approximation of the Term for Free exchanges

The next part of the proof clearly takes a tiny fraction of the informal proof as human reader perform quite well in understanding expectations and their summations.

In a formal treatment this has to be stated a bit more extensively.

We want to show the following inequation:

$E[map_{pmf}\ (\lambda x.\ \textsf{if}\ q \in set\ init$
$\quad \textsf{then if}\ (fst\ x)_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \sum j{<}k'.\ [\![fst\ x]\!](xs'_{[j]}){+}1$
$\quad \textsf{else}\ 0)\ D]$

$\leq 3\ /\ 4 * k$

If the requested element is not in the list, the left hand side gets 0 and thus the inequation is proven trivially.

In the case that the requested element is in the list, we first show how to transform the two terms individually by the following two lemmas and then combine them:

Transformation of the first term

Lemma 4.16. $(\sum x{\in}\{l \mid |l| = |init| \wedge l_{[q]}\}.\ k - k') = (k - k') * 2^{|init|\ -\ 1}$.

The first term is used iff $(fst\ x)_{[q]} = True$, hence we look at the sum over all bool lists of length $|init|$ with the qth bit fixed to *True*. As the inner sum is independent of the bool list of the outer sum it can be simplified to a product of $k - k'$ and the cardinality of the set. Obviously the latter equals to $2^{|init|\ -\ 1}$. Formalizing the last step includes proving the following lemma:

Lemma 4.17. *Let X and Y be disjoint sets of indices in the range* $[0, \ldots, m-1]$ *then,*
$|\{xs \mid (\forall\, i{\in}X.\ xs_{[i]}) \wedge (\forall\, i{\in}Y.\ \neg\ xs_{[i]}) \wedge |xs| = m\}| = 2^{m\ -\ |X|\ -\ |Y|}$

Transformation of the second term

Now we consider the more complex second sum:

Lemma 4.18. $(\sum x{\in}\{l \mid |l| = |init| \wedge \neg\ l_{[q]}\}.\ \sum j{<}k'.\ [\![x]\!](xs'_{[j]}){+}1) = 3\ /\ 2 * k' * 2^{|init|\ -\ 1}$

In the original term the second sum is used iff $(fst\ x)_{[q]} = False$, thus we look at the sum over all possible bit vectors that fix the bit on position q to *False*. Intuitively it is clear that there are $2^{|init|\ -\ 1}$ many. Furthermore all indices j in the inner sum are smaller than k' hence smaller than k and thus $xs'_{[j]} \neq q$. That means albeit fixing the qth position, all value combinations are still possible and the expectation for the inner expression is

$[\![x]\!](xs'_{[j]})+1 = 3 / 2$.

Although this might be a simple derivation for a mathematician, it is quite some work to formalize. In the heart of this formal proof lies the following lemma, which can be proven by an induction on the finite set S.

Lemma 4.19. *Let S, Tr and Fa be pairwise disjoint sets of indices in the range of $[0, \ldots, l-1]$, then the following equality holds:*

$(\sum x \in \{xs \mid (\forall i \in Tr.\ xs_{[i]}) \wedge (\forall i \in Fa.\ \neg\ xs_{[i]}) \wedge |xs| = l\}.\ \sum j \in S.\ [\![x]\!](j)+1) =$
$3 / 2 * |S| * 2^{l\ -\ |Tr|\ -\ |Fa|}$

Equational transformations to the goal

$$\begin{aligned}
& E[map_{pmf}\ (\lambda x.\ \textsf{if}\ (fst\ x)_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \textstyle\sum j{<}k'.\ [\![fst\ x]\!](xs'_{[j]})+1)\ D] \\
= \ & E[map_{pmf}\ (\lambda x.\ \textsf{if}\ x_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \textstyle\sum j{<}k'.\ [\![x]\!](xs'_{[j]})+1)\ (L\ |init|)] \\
= \ & \textstyle\sum x \in set_{pmf}\ (L\ |init|).\ (\textsf{if}\ x_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \sum j{<}k'.\ [\![x]\!](xs'_{[j]})+1) * pmf\ (L\ |init|)\ x \\
= \ & \textstyle\sum x \in \{l \mid |l| = |init|\}.\ (\textsf{if}\ x_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \sum j{<}k'.\ [\![x]\!](xs'_{[j]})+1) * (1 / 2)^{|init|} \\
= \ & (1 / 2)^{|init|} * (\textstyle\sum x \in \{l \mid |l| = |init|\}.\ \textsf{if}\ x_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \sum j{<}k'.\ [\![x]\!](xs'_{[j]})+1) \\
= \ & (1 / 2)^{|init|} * ((\textstyle\sum x \in \{l \mid |l| = |init| \wedge l_{[q]}\}.\ \textsf{if}\ x_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \sum j{<}k'.\ [\![x]\!](xs'_{[j]})+1) \\
& + (\textstyle\sum x \in \{l \mid |l| = |init| \wedge \neg\ l_{[q]}\}.\ \textsf{if}\ x_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \sum j{<}k'.\ [\![x]\!](xs'_{[j]})+1)) \\
= \ & (1 / 2)^{|init|} * ((\textstyle\sum x \in \{l \mid |l| = |init| \wedge l_{[q]}\}.\ k - k') + (\sum x \in \{l \mid |l| = |init| \wedge \neg\ l_{[q]}\}. \\
& \textstyle\sum j{<}k'.\ [\![x]\!](xs'_{[j]})+1)) \\
= \ & (1 / 2)^{|init|} * ((k - k') * 2^{|init|\ -\ 1} + 3 / 2 * k' * 2^{|init|\ -\ 1}) \\
= \ & 1 / 2 * (k - k' + k' * (3 / 2)) \\
\leq \ & 1 / 2 * (3 / 2 * (k - k') + k' * (3 / 2)) \\
= \ & 3 / 4 * k
\end{aligned}$$

With this in place we can transform the expectation. First we observe that the expression only depends on the first component of BIT's state, which is by Lemma 4.6 always $L\ |init|$. Then as $L\ |init|$ is finite we can rewrite the expectation as a finite sum over the expressions evaluated for a state times the probability of that state. By properties of L we know this probability is $(1 / 2)^{|init|}$ and the set of states are the *bool list*s with length $|init|$. We can pull out the factor and then divide the set of states on whether the requested element's bit is set or not. This decides which of the two sums is evaluated; the sum can be pulled apart and simplified. The next step is to use lemmas 4.16 and 4.18 and then contract the expression to $3 / 4 * k$.

Finally we obtain the result:

Lemma 4.20.

$$\begin{aligned}
& E[map_{pmf}\ (\lambda x.\ \textsf{if}\ q \in set\ init \\
& \qquad \textsf{then if}\ (fst\ x)_{[q]}\ \textsf{then}\ k - k'\ \textsf{else}\ \textstyle\sum j{<}k'.\ [\![fst\ x]\!](xs'_{[j]})+1 \\
& \qquad \textsf{else}\ 0)\ D] \\
& \leq 3 / 4 * k
\end{aligned}$$

4.4.3 Transformation of the Term for Paid Exchanges

A similar but far shorter development can be done for the term concerning the paid exchanges. We again observe that only the first component of the state is needed, then we sum over all the possible internal states. The property of *gebub n i* being in bounds of the internal state is needed to complete the proof, and we obtain:

Lemma 4.21. $E[map_{pmf}\ (\lambda x.\ \sum i{<}|paid_{A[n]}|.\ [\![fst\ x]\!](gebub\ n\ i){+}1)\ D] = 3\ /\ 2 * |paid_{A[n]}|$

4.4.4 Combine the Results

$$
\begin{aligned}
& t_{BIT}\ n + \Phi\ (n+1) - \Phi\ n \\
= \ & E[map_{pmf}\ (\lambda x.\ cost\ x + \Phi_2\ x - \Phi_0\ x)\ D] \\
= \ & E[map_{pmf}\ (\lambda x.\ k+1)\ D] + \\
& (E[map_{pmf}\ (\lambda x.\ \textsf{if}\ q \in set\ init \\
& \qquad \textsf{then if}\ (fst\ x)_{[q]}\ \textsf{then}\ k - k'\ \textsf{else} \sum j{<}k'.\ [\![fst\ x]\!](xs'_{[j]}){+}1 \\
& \qquad \textsf{else}\ 0)\ D] + \\
& E[map_{pmf}\ (\lambda x.\ \sum i{<}|paid_{A[n]}|.\ [\![fst\ x]\!](gebub\ n\ i){+}1)\ D]) \\
\leq \ & k + 1 + (3\ /\ 4 * k + 3\ /\ 2 * |paid_{A[n]}|) \\
= \ & 7\ /\ 4 * k + 3\ /\ 2 * |paid_{A[n]}| + 1 \\
\leq \ & 7\ /\ 4 * (k + |paid_{A[n]}|) + 1 \\
= \ & 7\ /\ 4 * (t_A\ n - 1) + 1 \\
= \ & 7\ /\ 4 * t_A\ n - 3\ /\ 4
\end{aligned}
$$

First we use Equation 4.1. Then with monotonicity of E, we apply the approximation of the expression inside (Lemma 4.12) and pull out the sums with linearity of E. Then we use the results from Lemma 4.20 and 4.21 to obtain concrete bounds for the expectations. After some transformation we can plug in the cost of A and then simplify to the desired result.

Finally we finish the proof of the main lemma.

□

4.5 Lift the Result to the Whole Request List

The result can be lifted to $T_{BIT}\ n$ with the potential function method:

Corollary 4.22. $n \leq |qs| \Longrightarrow T_{BIT}\ n \leq 7\ /\ 4 * T_A\ n - 3\ /\ 4 * n$

Furthermore, recall that we first filtered out all paid exchanges from algorithm $\bar{A}$ in order to obtain a well-formed algorithm A, we already showed that this filtering does not alter the effect of the algorithm but decreases its cost. As we showed the inequality for the filtered algorithm A we can simply obtain the corresponding result for $\bar{A}$:

Corollary 4.23 ([9, Theorem 2.1])**.** $n \leq |qs| \Longrightarrow T_{BIT}\ n \leq 7\ /\ 4 * T_{\bar{A}}\ n - 3\ /\ 4 * n$

With an upper bound for the cost of BIT ($T_{BIT}\ n \leq n * |init|$) we obtain the competitiveness result:

Lemma 4.24.
$[\![n \leq |qs|; init \neq []; \forall i{<}n.\ qs_{[i]} \in set\ init]\!] \Longrightarrow T_{BIT}\ n \leq (7\ /\ 4 - 3\ /\ (4 * |init|)) * T_{\bar{A}}\ n$

Until now we used the newly defined functions (c.f. Section 4.2.1) to express the costs of BIT and $\bar{A}$; in order to integrate them into our framework for competitive analysis we have to show the equivalence with the related notions:

Lemma 4.25. $T_{BIT}\ |qs| = T_{on}\ BIT\ qs\ init$
$T_{\bar{A}}\ |qs| = T\ init\ qs\ acts$

Ultimately we can state the final lemma of this section: BIT is $\frac{7}{4}$-competitive against any strategy A:

Lemma 4.26. $[\![init \neq []; \forall i{<}|qs|.\ qs_{[i]} \in set\ init]\!] \Longrightarrow T_{on}\ BIT\ qs\ init \leq (7\ /\ 4 - 3\ /\ (4 * |init|))$ $*\ T\ init\ qs\ acts$

4.6 Generalize Competitivness of BIT

It obviously follows that BIT is also $\frac{7}{4}$-competitive against the optimal strategy. Formally this is done by interpreting the above locale with any valid strategy and conclude from the interpretation of Lemma 4.26 that BIT is also competitive against the optimal strategy:

Theorem 4.27. $compet\ BIT\ (7\ /\ 4)\ \{init\ |\ init \neq [] \wedge set\ init = \{0..{<}|init|\}\}$

4.7 Conclusion and Remarks

With the proof of competitivenss of BIT, we have seen a similar proof as for MTF: we fixed an adversary and a specific request sequence (via a locale) and then showed competitivness in that setting by an amortized analysis. Again, the potential function technique was used, with a slightly more complicated potential function. As we chose the adversary and the request sequence arbitrarily we were able to conclude that BIT is 1.75-competitive. As BIT is a random algorithm we had to express statements about distributions of states which complicated the proof. Eventually, reasoning about simple probability distributions required more effort than its informal counterpart: concepts that mathematicians are quite familiar with – e.g. uniform distributions of bit vectors, reasoning with expectations and transformations of sums – had to be formalized and thus blew up the proof.
Note that surprisingly BIT's behavioural brother Random Move To Front (RMTF), which moves any requested element to front with probability $\frac{1}{2}$, is no better than 2-competitive [9, Section 2.3].

As we have seen in the last two chapters, proofs for competitive analysis of quite easy algorithms already are quite complicated and intricate. But we also have seen that most arguments involve only the relationship between two elements: i.e. we counted the inversions of two elements. In the following chapter we will introduce the list factoring technique, which allows us for a specific class of algorithms (including MTF, BIT and others)

to first reason only on lists of length 2 and then naturally lift the result to lists of arbitrary list length. Typically these algorithms are much easier to analyse on the restricted lists.

5 List factoring technique

In the last two chapters we have seen proofs for competitiveness of the algorithms MTF and BIT. Albeit these algorithms are simple to state, their analysis is already quite complicated. In order to analyse more complex algorithms we long for better techniques.

The proof technique *list factoring* enables us to reason about a certain algorithm only on lists of length 2 and then lift the result to lists of arbitrary length. As most algorithms collapse into quite easy ones, once they only work on two elements, the proofs typically get much shorter, and thus enable us to tackle more involved algorithms.

Borodin gives quite easy proves of TS being 2-comp, BIT being 1.75-comp and their combination COMB being 1.6-comp, once the proof technique of list factoring is available.

The downside of this approach is, that a lot of work has to be done in order to obtain this proof technique.

In this chapter we introduce the list factoring technique for analyzing algorithms for the list update problem. Therefor we first present a different representation of the cost of a list update algorithm with which we can decompose this cost to the costs only involving pairs of elements. We then introduce the pairwise property of online algorithms, which is satisfied by a number of proposed algorithms (e.g BIT, MTF, TS, etc.). With these two ingredients we are able to show the factoring lemma which enables us to lift competitiveness results of lists of length two to arbitrary list lengths.

Note that from this chapter on we consider the partial cost model for the list update problem, i.e. an access to the front element has cost 0.

5.1 Another view on the cost of an algorithm for the list update problem

The list factoring technique only works for algorithms that do not execute paid exchanges. These have the property that a request's cost only depends on the position in the list.

The main idea of the list factoring technique is to count the cost of accesses in a different way: Instead of thinking about the cost of a request as the position i in the list and attributing the entire access cost of i to that element, we describe it as the number of elements that precede the requested element. We thus change our view and attribute a "blocking cost" of 1 to every element that precedes the requested element. For the requested element and all following the "blocking cost" is 0.

Formally we state the blocking cost of an element x for the requested element $qs_{[i]}$ for a current state tuple s as. Remember that a state tuple (is, c) is a pair of an internal state is and a list configuration c.

Definition 5.1. $ALG\ x\ qs\ i\ s = (\textsf{if}\ x < qs_{[i]}\ in\ snd\ s\ \textsf{then}\ 1\ \textsf{else}\ 0)$

We now lift this definition into the randomized world, where we have to cope with a distribution over states and expectations: *ALG′ A qs init i x* determines the expected blocking cost of element x in the ith step of the execution of the online algorithm A on the request sequence *qs* starting from initial list state *init*.

Definition 5.2. $ALG'\, A\ qs\ init\ i\ x = E[map_{pmf}\ (\lambda xa.\ ALG\ x\ qs\ i\ xa)\ (config^*\ A\ qs\ init\ i)]$

We can find another representation of the cost of an online algorithm without paid exchanges:

$$
\begin{aligned}
& T^*_{on}\ (I, S)\ qs\ init \\
= \ & \textstyle\sum i{<}|qs|.\ t^*_{on}\ (I, S)\ qs\ init\ i \\
= \ & \textstyle\sum i{<}|qs|.\ \sum x{\in}set\ init.\ ALG'\ (I, S)\ qs\ init\ i\ x \\
= \ & \textstyle\sum x{\in}set\ init.\ \sum i{<}|qs|.\ ALG'\ (I, S)\ qs\ init\ i\ x \\
= \ & \textstyle\sum x{\in}set\ init.\ \sum y{\in}set\ init.\ \sum i \mid i < |qs| \wedge qs_{[i]} = y.\ ALG'\ (I, S)\ qs\ init\ i\ x \\
= \ & \textstyle\sum (x, y){\in}\{(x, y) \mid x \in set\ init \wedge \\
& \qquad\qquad\qquad y \in set\ init \wedge \\
& \qquad\qquad\qquad x \neq y\}.\ \sum i \mid i < |qs| \wedge qs_{[i]} = y.\ ALG'\ (I, S)\ qs\ init\ i\ x \\
= \ & \textstyle\sum (x, y){\in}\{(x, y) \mid x \in set\ init \wedge \\
& \qquad\qquad\qquad y \in set\ init \wedge \\
& \qquad\qquad\qquad x < y\}.\ \sum i \mid i < |qs| \wedge (qs_{[i]} = y \vee qs_{[i]} = x). \\
& \qquad\qquad\qquad\qquad\qquad ALG'\ (I, S)\ qs\ init\ i\ y\ + \\
& \qquad\qquad\qquad\qquad\qquad ALG'\ (I, S)\ qs\ init\ i\ x
\end{aligned}
$$

First we unfold the definition of the algorithm's cost, then the cost of step i is equivalent to the sum of blocking cost of all elements in step i. We rearrange the summations and denote the inner summation of the last expression *ALGxy A qs init x y*, meaning the cost generated by x blocking y or vice versa:

Definition 5.3.
$ALGxy\ A\ qs\ init\ x\ y = (\sum i \mid i < |qs| \wedge qs_{[i]} \in \{y, x\}.\ ALG'\, A\ qs\ init\ i\ y + ALG'\, A\ qs\ init\ i\ x)$

We can summarize the above derivation:

Lemma 5.4 ([9, Equation 1.4])**.**
$T^*_{on}\ (I, S)\ qs\ init = (\sum (x, y){\in}\{(x, y) \mid x \in set\ init \wedge y \in set\ init \wedge x < y\}.\ ALGxy\ (I, S)\ qs\ init\ x\ y)$

5.1.1 The pairwise property

At this point we want to find a possibility to determine *ALGxy A qs init x y*. The only thing the term depends on is the relative order of x and y during the execution. Note that this order can only change when either x or y is requested and thus can get in front of the other element via free exchanges.

We now examine the cost of an algorithm on a projected list and request sequence:

Denote with $qs^{[\{x,\ y\}]}$ the projection of qs over x and y, being the request sequence qs after deleting all requests for elements other than x and y. Similarly let $init^{[\{x,\ y\}]}$ be the projection of the initial list.

Thus we can state the cost of serving the projected request sequence on the projected initial list as:

$T^*_{on}\ A\ qs^{[\{x,\ y\}]}\ init^{[\{x,\ y\}]}$

Definition 5.5 (pairwise property)**.** *We then say that the algorithm A satisfies the* pairwise property *if*

$$\begin{aligned}&\forall\, qs\ init.\\&\quad \forall\,(x, y){\in}\{(x, y) \mid x \in set\ init \wedge y \in set\ init \wedge x < y\}.\\&\qquad T^*_{on}\ A\ qs^{[\{x,\ y\}]}\ init^{[\{x,\ y\}]} = ALGxy\ A\ qs\ init\ x\ y\end{aligned}$$

Remark: Algorithm MTF and BIT are examples of algorithms that satisfy the pairwise property. Also algorithms TS and COMB satisfy it.

5.1.2 Desire for the list factoring technique

With Lemma 5.4 and the definition of the pairwise property we are in the position to describe the list factoring technique:

Suppose we have an algorithm A that does not use paid exchanges and satisfies the pairwise property. Assume for the moment that OPT also satisfies the pairwise property as well as Lemma 5.4. Now suppose that we have proven that A is c-competitive for all projected request sequences $qs^{[\{x,\ y\}]}$ and initial lists $init^{[\{x,\ y\}]}$:

$T^*_{on}\ A\ qs_2\ init_2 \leq c * T^*_{on}\ OPT\ qs^{[\{x,\ y\}]}\ init^{[\{x,\ y\}]}$

With the pairwise property of both A and OPT we obtain

$ALGxy\ A\ qs\ init\ x\ y \leq c * ALGxy\ OPT\ qs\ init\ x\ y$

By Lemma 5.4 we could conclude that A is c-competitive.

$$\begin{aligned}&\quad T^*_{on}\ A\ qs\ init\\&= \textstyle\sum\,(x, y){\in}\{(x, y) \mid x \in set\ init \wedge y \in set\ init \wedge x < y\}.\ ALGxy\ A\ qs\ init\ x\ y\\&\leq \textstyle\sum\,(x, y){\in}\{(x, y) \mid x \in set\ init \wedge y \in set\ init \wedge x < y\}.\ c * ALGxy\ OPT\ qs\ init\ x\ y\\&= c * T^*_{on}\ OPT\ qs\ init\end{aligned}$$

Unfortunately, OPT neither can avoid paid exchanges in general nor does it necessarily satisfy the pairwise property. That is why some detour has to be taken. In the next section we develop similar equations to Lemma 5.4 and the pairwise property for the optimal offline algorithms.

5.2 List Factoring for OPT

The crucial lack, why we cannot conduct the same development for OPT as in Lemma 5.4 is that OPT may use paid exchanges. Thus we have to take these into account.

For that purpose we define the function *ALG-P sws x y s* that determines how often elements x and y are swapped while executing the swaps *sws* on the list s.

Note that we now want to use list factoring for a specific strategy (say *Strat*), thus we do not have to talk about expectations. So we can easily lift *ALG-P* up to *ALG-Pxy Strat qs init x y* – denoting the number of paid exchanges between elements x and y while executing *Strat* on request sequence *qs* and initial list *init*. Similarly we lift the blocking cost *ALG* to *ALGxy-det*.

Then we are able to state the following theorem:

Theorem 5.6 ([9, Equation 1.7]). *Suppose we have a strategy Strat that attains the optimal cost on qs and init, then the optimal cost for the projected case is at most the blocking cost plus the number of paid exchanges executed between x and y:*
$T^*_{opt}\ init^{[\{x,\ y\}]}\ qs^{[\{x,\ y\}]} \leq \textit{ALGxy-det Strat qs init x y} + \textit{ALG-Pxy Strat qs init x y}$

Proof. Note that the right-hand side of this inequality gives the total cost of some offline algorithm $Strat^{\{x,\ y\}}$ that is a projection of *Strat* over x and y: It includes all costs incurred by *Strat* for either accesses (via the blocking costs) and paid exchanges between x and y. The proof can be established by constructing $Strat^{\{x,\ y\}}$ and showing that its total cost in serving $qs^{[\{x,\ y\}]}$ is the right-hand side of the inequality. Surely this algorithm pays at least as much as the optimal offline algorithm. ☐

Furthermore, with a similar development as in Lemma 5.4, taking into account the paid exchanges, we can prove:

Theorem 5.7 ([9, Equation 1.8]).
$T^*\ \textit{init qs Strat} =$
$(\sum (x, y){\in}\{(x, y) \mid x \in \textit{set init} \wedge$
$y \in \textit{set init} \wedge$
$x < y\}.\ \textit{ALGxy-det Strat qs init x y} +$
$\textit{ALG-Pxy Strat qs init x y})$

Combining the last two theorems we can conclude:

Corollary 5.8. $(\sum (x, y){\in}\{(x, y) \mid x \in \textit{set init} \wedge y \in \textit{set init} \wedge x < y\}.\ T^*_{opt}\ init^{[\{x,\ y\}]}\ qs^{[\{x,\ y\}]})$
$\leq T^*_{opt}\ \textit{init qs}$

5.3 Factoring Lemma

Now as we have taken this detour, with the help of the pairwise property, Lemma 5.4 and Corollary 5.8 we can easily show the desired result:

Theorem 5.9 ([9, Lemma 1.2]). *Assume α to be nonnegative, c to be greater than 1 and A to be an online algorithm that has the pairwise property. If A is c-competitive on lists of length 2*
pairwise A
we can conclude that A is c-competitive on lists of arbitrary list length:
$\forall\, s0 \in S0.\ \exists\, b \geq 0.\ \forall\, qs.\ T^{*}_{on}\ A\ qs\ s0 \leq c * T^{*}_{opt}\ s0\ qs + b.$

6 OPT2: an Optimal Algorithm for Lists of Length 2

One key feature of the list factoring technique, introduced in the last chapter, is certainly that even complicated algorithms for the list update problem are surprisingly simple on lists of length 2. It is even possible to state an optimal offline algorithm: OPT2.

We have seen in the analysis of MTF and BIT, that in principle competitive analysis can be carried out without any knowledge of the optimal offline algorithm. With OPT2 we have more information about the structure of the optimal algorithm which can be used when proving competitivness of algorithms for the list update problem on lists of size 2.

In this chapter we study the nature of OPT2: we give its definition due to Reingold and Westbrook [20], show that it indeed attains the optimal cost on lists of size 2 and then determine the cost of OPT2 on different specific request sequences.

6.1 Formalization of OPT2

First we state the informal definition due to Reingold and Westbrook [20]:

Definition 6.1 (OPT2 informal)**.** *After each request, move the requested item to the front via free exchanges if the next request is also to that item. Otherwise do nothing.*

Observe that this algorithm only needs knowledge of the current and next request. Thus OPT2 is neither a pure offline nor an online algorithm; it is called an algorithm with lookahead. Further remarks on such algorithms can be found in the last section of this chapter.

We define a function OPT2 that, given a request sequence and an initial list of two elements, generates OPT2's strategy:

Definition 6.2 (OPT2)**.**

$$
\begin{aligned}
&\mathit{OPT2}\ []\ [x, y] &&= []\\
&\mathit{OPT2}\ [a]\ [x, y] &&= [(0, [])]\\
&\mathit{OPT2}\ (a \cdot b \cdot \sigma')\ [x, y] &&= \textsf{if}\ a = x\ \textsf{then}\ (0, []) \cdot \mathit{OPT2}\ (b \cdot \sigma')\ [x, y]\\
& &&\quad \textsf{else if}\ b = x\ \textsf{then}\ (0, []) \cdot \mathit{OPT2}\ (b \cdot \sigma')\ [x, y]\\
& &&\quad\quad \textsf{else}\ (1, []) \cdot \mathit{OPT2}\ (b \cdot \sigma')\ [y, x]
\end{aligned}
$$

Two simple properties of OPT2 can be stated: The length of the strategy matches the length of the request sequence and if the next requested element is in front of the list OPT2 will not do anything in this step. Recall that an action consists of the number of positions the requested element is moved forward by free exchanges and a list of indices for the paid exchanges.

Lemma 6.3. $|OPT2\ \sigma\ [x, y]| = |\sigma|$
$OPT2\ (x \cdot \sigma')\ [x, y] = (0, []) \cdot OPT2\ \sigma'\ [x, y]$

6.2 OPT2 is Optimal on Lists of Length 2

Before we tackle the proof for OPT2 observe that the following lemma holds:

Lemma 6.4. $T^*_{opt}\ [x, y]\ \sigma \leq 1 + T^*_{opt}\ [y, x]\ \sigma$

Proof. Suppose we have a strategy S to optimally serve the request sequence σ with initial list $[y, x]$ and look for a strategy to serve σ starting from $[x, y]$. We can first swap the two elements by one paid exchange and then use S to serve the sequence. This way we can bound the optimal cost for σ starting from $[x, y]$. ☐

Let us now turn to proving OPT2's optimality. As OPT2 obviously pays at least as much as the optimal offline algorithm the more demanding part of the proof is showing that OPT2 pays no more than the optimal offline algorithm:

Lemma 6.5 ([20, Proposition 4])**.** $T^*\ [x, y]\ \sigma\ (OPT2\ \sigma\ [x, y]) \leq T^*_{opt}\ [x, y]\ \sigma$

Proof. First we will give an outline of the proof and then show the details for one case.

Assume the initial list to be $[x, y]$ and only elements x and y being requested.

The lemma is proven by well-founded induction on the length of the request sequence σ.

We start by doing a case distinction on whether the requested element is x and thus in front of the list.

If indeed x is requested, no matter what requests follow, OPT2 will do nothing in this move and keep the list untouched (Lemma 6.3). The optimal offline algorithm now has two options: Either it acts as OPT2 – then the induction hypothesis can be applied right away – or it swaps the list, then we can use Lemma 6.4 and it remains to show that serving the request while reversing the list costs at least 1 more than acting like OPT2 and leaving the list untouched.

In the second case (the requested element is y) another case distinction has to be done on the second requested element. Depending on whether the optimal algorithm transforms the list as OPT2 or not, the induction hypothesis can be applied at once or again with lemma 6.4.

We will present the first case in more detail. As for the second case, the proof is established essentially by the same method, only some more case distinctions obscure the argument. For more details feel free to look into the proof text.

Consider now the case that the request sequence is of the form $\sigma = x \cdot \sigma'$. Our current goal is $T^*\ [x, y]\ (x \cdot \sigma')\ (OPT2\ (x \cdot \sigma')\ [x, y]) \leq T^*_{opt}\ [x, y]\ (x \cdot \sigma')$.

To show that, we fix an arbitrary optimal strategy *Strat*.

In the case that *Strat* leaves the list untouched ($step\ [x, y]\ x\ (hd\ Strat) = [x, y]$) the induction hypothesis suffices to complete the proof:

$$
\begin{aligned}
& T^* \; [x, y] \; (x \cdot \sigma') \; (OPT2 \; (x \cdot \sigma') \; [x, y]) \\
= \;& T^* \; (step \; [x, y] \; x \; (hd \; (OPT2 \; (x \cdot \sigma') \; [x, y]))) \; \sigma' \\
& \; (OPT2 \; \sigma' \; (step \; [x, y] \; x \; (hd \; (OPT2 \; (x \cdot \sigma') \; [x, y])))) \\
= \;& T^* \; [x, y] \; \sigma' \; (OPT2 \; \sigma' \; [x, y]) \\
\leq \;& T^*_{opt} \; [x, y] \; \sigma' \\
\leq \;& T^* \; [x, y] \; \sigma' \; (tl \; Strat) \\
\leq \;& t^* \; [x, y] \; x \; (hd \; Strat) + T^* \; (step \; [x, y] \; x \; (hd \; Strat)) \; \sigma' \; (tl \; Strat) \\
= \;& T^* \; [x, y] \; (x \cdot \sigma') \; Strat
\end{aligned}
$$

First we unfold the calculation of OPT2's strategy: it will not do anything and also pay nothing, as x is located in the first position. Then we can apply the induction hypothesis for the shorter request sequence σ'. Obviously, if we use the tail of *Strat* as a strategy for the request sequence σ' it will cost at least as much as the optimal strategy. The last two steps bridge the gap to the total cost of strategy *Strat* on the whole request sequence and are justified by the fact that *Strat* does not change the list in this case and the cost of that step is non-negative.

The case that *Strat* reverses the list ($step \; [x, y] \; x \; (hd \; Strat) = [y, x]$) is a bit more intricate as we cannot simply apply the induction hypothesis. Lemma 6.4 comes to the rescue:

$$
\begin{aligned}
& T^* \; [x, y] \; (x \cdot \sigma') \; (OPT2 \; (x \cdot \sigma') \; [x, y]) \\
= \;& T^* \; (step \; [x, y] \; x \; (hd \; (OPT2 \; (x \cdot \sigma') \; [x, y]))) \; \sigma' \\
& \; (OPT2 \; \sigma' \; (step \; [x, y] \; x \; (hd \; (OPT2 \; (x \cdot \sigma') \; [x, y])))) \\
= \;& T^* \; [x, y] \; \sigma' \; (OPT2 \; \sigma' \; [x, y]) \\
\leq \;& T^*_{opt} \; [x, y] \; \sigma' \\
\leq \;& 1 + T^*_{opt} \; [y, x] \; \sigma' \\
\leq \;& 1 + T^* \; [y, x] \; \sigma' \; (tl \; Strat) \\
= \;& 1 + T^* \; (step \; [x, y] \; x \; (hd \; Strat)) \; \sigma' \; (tl \; Strat) \\
\leq \;& t^* \; [x, y] \; x \; (hd \; Strat) + T^* \; (step \; [x, y] \; x \; (hd \; Strat)) \; \sigma' \; (tl \; Strat) \\
= \;& T^* \; [x, y] \; (x \cdot \sigma') \; Strat
\end{aligned}
$$

We begin as for the first case, but after applying the induction hypothesis we use Lemma 6.4. Then again using *Strat* on the remainder of the request sequence costs at least as much as the optimal strategy and we bridge the gap until we obtain the total cost of *Strat* on the request sequence. The tricky part, showing that $1 \leq t^* \; [x, y] \; x \; (hd \; Strat)$, can be done by a proof by contradiction: if the strategy does not pay anything for the serving of the request, it must not do any paid exchange but then it is not able to reverse the list.

□

As a corollary we obtain the result that OPT2 is optimal on lists of length 2.

Corollary 6.6 (OPT2 is optimal)**.** $T^* \; [x, y] \; qs \; (OPT2 \; qs \; [x, y]) = T^*_{opt} \; [x, y] \; qs$

With OPT2 we obtain an reference algorithm which can be used to analyse online algorithms on lists of length 2. In the following chapter we analyze the online algorithm TS and determine its cost on 3 different types of request sequences for lists of size 2. With the knowledge of the optimal cost for these request sequences we can prove TS to be 2-competitive. With OPT2 at hand this can easily be determined.

6.3 Further properties of OPT2

In this section we will examine further properties of OPT2 that will help in the analysis of other online algorithms.

First, for known request sequences it is now easy to calculate the optimal cost on lists of length 2. We can use this to determine the cost of certain classes of request sequences which can be represented by regular expressions over two elements x and y. The proofs can be established by mere simulation and induction on the number of repetitions in the case of star expressions.

σ	$T^*\ [x,y]\ \sigma\ (OPT2\ \sigma\ [x,y])$
$x^?yy$	1
$x^?yx(yx)^*yy$	$\frac{\lvert\sigma\rvert}{2}$
$x^?yx(yx)^*x$	$\frac{\lvert\sigma\rvert - 1}{2}$

Table 6.1: Costs of OPT2 for request sequence of three classes.

Furthermore we can give an easy upper bound on the cost of OPT2: Obviously OPT2 can at most have cost 1 for every request and we obtain the following lemma:

Lemma 6.7. *$T^*\ [x,y]\ \sigma\ (OPT2\ \sigma\ [x,y]) \leq |\sigma|$*

Moreover we can show that requesting the last requested element again has additional cost at most 1.

Lemma 6.8.
If $R \in \{[x,y], [y,x]\}$ then $T^*\ R\ (\sigma\ @\ [x,x])\ (OPT2\ (\sigma\ @\ [x,x])\ R) \leq T^*\ R\ (\sigma\ @\ [x])\ (OPT2\ (\sigma\ @\ [x])\ R) + 1$

Proof. Unfortunately the calculation of OPT2's strategy depends on the rest of the request sequence (if only it needs lookahead 1) thus one cannot append one more element easily. One way of proving the lemma nonetheless is by induction on σ for arbitrary R and again a lot of case distinctions which can fortunately be solved mechanically. □

Finally, observe that in an execution of OPT2 after the occurrence of two equal elements xx one after the other OPT2's list has element x at the front. This enables us to partition the request sequence in phases and restart OPT2 repeatedly.

Lemma 6.9. *If $R \in \{[x,y], [y,x]\}$ then $OPT2\ (\sigma_1\ @\ [x,x]\ @\ \sigma_2)\ R = OPT2\ (\sigma_1\ @\ [x,x])\ R\ @\ OPT2\ \sigma_2\ [x,y]$.*

Proof. This lemma can be proven by an induction on σ_1 for arbitrary R. For the inductive step assume without loss of generality that $R = [x,y]$; then with a case distinction on the following requests one can apply the induction hypothesis. □

6.4 Remarks

As we observed, OPT2 is no online algorithm but an algorithm with lookahead.

Definition 6.10 ([20]). *A list update algorithm is said to have* lookahead-$k(n)$ *if it makes each decision knowing only the next* $k(n)$ *unprocessed requests, where* $k(n)$ *is some function independent of the request sequence, but perhaps depending on the initial list size.*

Consequently an online algorithm has lookahead-0, while an offline algorithm has unbounded lookahead. For further results on algorithms for the list update problem with lookahead [1] by Albers is a good starting point.

In the next chapter we will exploit the knowledge we obtained in this chapter when analysing the online algorithm TS. As we have seen we can partition the request sequence in phases ending with two requests to the same element, knowing OPT2's list state at the end of these phases. This technique will be introduced in the next chapter as *phase partitioning* and will be used together with the list factoring technique to show TS being 2-competitive.

7 TS: another 2-competitve Algorithm

We now study another deterministic algorithm: TS was introduced as a member of the family of randomized online algorithms TIMESTAMP due to Albers [2]. The best member of this family is Φ-competitive (with $\Phi \approx 1.62$ being the golden ratio). A little later a combination of TS and BIT called COMB was presented by Albers et al [4] and it even improved that bound to 1.6.

This is to date the best known algorithm for the list update problem. As for a lower bound of the competitive factor, Ambühl showed that no randomized online algorithm (in the partial cost model) can attain a better competitive ratio than 1.50155 [5].

The proof of the competitivness of COMB in [4] is based on the analysis of TS and BIT on lists of length 2. The first step of this development, proving TS's 2-competitiveness for lists of length 2, is the topic of this chapter.

In the following we will first define the deterministic online algorithm TS, then will introduce to the phase partitioning technique that we suggested in the last chapter and finally use it to show that TS is 2-competitive on lists of length 2. Assuming we had proven the pairwise property of TS we could use the list factoring lemma to conclude TS being 2-competitive on lists of arbitrary length.

7.1 Definition of TS

The deterministic online algorithm due to Albers [2] can be formulated as follows:

Definition 7.1 (TS informal)**.** *After each request, the accessed item x is inserted immediately in front of the first item y that precedes x in the list and was requested at most once since the last request to x. If there is no such item y or if x is requested for the first time, then the position of x remains unchanged.*

In order to formalize this algorithm we take the history of the request sequences already processed as the internal state. Consequently we initialize TS with an empty history list. Also we formalize the deterministic step.

Definition 7.2.
TS-init s $=$ *return*$_{pmf}$ [] *TS-step-d is s q* $=$
$(($**let** *li* $=$ *index is q*
  in if *li* $= |is|$ **then** *0*
    else let *sincelast* $=$ *take li is*;
        $S = \{x \mid x < q$ *in s* $\wedge$ *count sincelast x* $\leq 1\}$
      in if $S = \emptyset$ **then** *0* **else** *index s q* $-$ *Min* $($*index s* ' $S)$,

$[]),$
$q \cdot is)$

To determine the action taken by TS to serve a request q with list state being s and history being is, we first compute to what position the requested element is moved, then add no paid exchanges and update the history with the currently requested element q. As for the position q gets moved to, it is a literal translation of the informal definition. Note that *take n xs* returns the length n prefix of xs.

To conform with the format of the framework we lift this definition to an online algorithm:

Definition 7.3.
$\textit{TS-step is s q} = \textit{return}_{pmf}\ (\textit{TS-step-d is s q})$
$\textit{rTS} = (\textit{TS-init}, \textit{TS-step})$

As we want to avoid working in the randomized setting with expectations and distributions we first work on a deterministic version while establishing the proof and will later reintegrate it into the framework. Hence we define a deterministic *TSstep* s.t. the following lemma holds:

Lemma 7.4. $\textit{set}_{pmf}\ (\textit{config}^*\ \textit{rTS qs init n}) = \{\textit{TSdet init}\ []\ \textit{qs n}\}$

We denote by *s-TS init initH qs n* the list state of TS after serving the nth request of the request sequence qs starting with the initial list *init* and initial history list *initH*. Thus *s-TS* is just the first component of *TSdet*.
As TS does not use paid exchanges the cost for a single step only depends on the position of the requested element. Hence the costs for a single step and the whole execution are

Lemma 7.5.
$t^*_{TS}\ \textit{init initH qs n} = \textit{index}\ (\textit{s-TS init initH qs n})\ qs_{[n]}$
$T^*_{TS}\ \textit{init initH qs} = (\sum i{<}|qs|.\ t^*_{TS}\ \textit{init initH qs i})$

and it can be easily shown that we can reintegrate it into the framework:

Lemma 7.6. $T^*_{on}\ \textit{rTS qs init} = T^*_{TS}\ \textit{init}\ []\ qs$

From that point on we will work in the deterministic domain, being able to lift it back into the framework with lemma 7.6.

Note that TS maintains the history list the expected way:

Lemma 7.7. $n \leq |xs| \Longrightarrow \textit{fst}\ (\textit{TSdet init initH}\ (xs\ @\ zs)\ n) = \textit{rev}\ (\textit{take}\ n\ (xs\ @\ zs))\ @\ \textit{initH}$

7.2 Phase Partitioning Technique

In this technique we cut the request sequence in phases ending with two consecutive requests to the same element (*xx* or *yy*).

These phases come in different types, which will be characterised by regular expressions over the alphabet $\{x, y\}$. In order to show equivalence of the languages represented by these, we use the theory of regular expressions: A regular expression equivalence checker has already been formalized in Isabelle/HOL [18], which we extend to regular expressions with variables.

With this tool in place, we can easily show that the regular expressions defining the different types really cover all possible request sequences.

7.2.1 Regular Expressions Equivalence

In the following we will partition all request sequences into phases that end with two consecutive requests to the same element and possibly a trailing incomplete phase. These phases can be summarized into 4 types, which can be represented by regular expressions over the elements x and y. In order to check whether we covered all possible request sequences we formalize these regular expressions and use a regular expression equivalence checker available in Isabelle/HOL.

All request sequences can be described by $(x+y)^*$, the phases that end in two identical elements by $x^?(yx)^*yy + y^?(xy)^*xx$ and the incomplete phases (sequences that do not contain any occurrence of two consecutive elements) by $x^?(yx)^*y + y^?(xy)^*x$.

First we show that any request sequence can be described by a concatenation of several phases and an trailing incomplete phase:

Lemma 7.8. $lang\ (myUNIV\ x\ y) = lang\ ((x^?(yx)^*yy + y^?(xy)^*xx)^*(1 + x^?(yx)^*y + y^?(xy)^*x))$

Furthermore a phase can be of one of the following types:

	σ
A	$x^?yy$
B	$x^?yx(yx)^*yy$
C	$x^?yx(yx)^*x$
D	xx

Table 7.1: 4 types of phases.

In Chapter 6 we determined OPT2's cost for serving request sequences of the first three types. We verify that they cover all proper phases:

Lemma 7.9. $lang\ (x^?yy + x^?yx(yx)^*yy + x^?yx(yx)^*x + xx) = lang\ (x^?(yx)^*yy + y^?(xy)^*xx)$

Note that these results naturally also hold for x and y interchanged. All the proofs of this section can be proven mechanically with the regular expression equivalence checker.

Example 7.10. *Figure 7.10 shows the request sequence xyxyxxxyxyxyxyyxxxyxyxyxyxyxy partitioned into phases and a trailing incomplete phase. Any such incomplete phase can be padded to form a proper phase by repeating the last requested element. This also can be ensured via proving the equivalence of two regular expressions.*

σ:	xyxyxx	xyxyxyxyy	xx	xyxyxyxyxyxy	y
type:	C	B	D	incomplete	padding

7.2.2 Analysis of the Phases

Let us first consider the 4 different phase types we just introduced and proof that TS is 2-competitive for request sequences of these types:

Without loss of generality, we only consider servings of the phases with the element x in front of the current list. For every phase we assume that a certain invariant holds before the serving of the sequence: $h = [] \vee (\exists hs.\ h = [x, x] @ hs)$

It states that either we are in the beginning of a request sequence and no element has been processed so far or the last two requests went to element x. This invariant implies that for the first request to y, the element would not be moved to the front of x (c.f. definition 7.2).

We now can show:

	σ	$T^*_{TS}\ [x, y]\ h\ \sigma$	$T^*\ [x, y]\ \sigma\ (OPT2\ \sigma\ [x, y])$
A	$x^? yy$	2	1
B	$x^? yx(yx)^* yy$	$2 * \frac{\lvert\sigma\rvert - 3}{2}$	$\frac{\lvert\sigma\rvert}{2}$
C	$x^? yx(yx)^* x$	$2 * \frac{\lvert\sigma\rvert - 3}{2}$	$\frac{\lvert\sigma\rvert - 1}{2}$
D	xx	0	0

Table 7.2: Costs of TS for request sequence of the 4 types.

Table 7.2.2 shows the costs of TS and OPT2 for the 4 respective types. We now verify the numbers for TS:

For $x^? yy$, we first may have a request to x that costs nothing as x is in the front of the list. The first request to y costs one and the element will not be moved to the front, as the invariant still holds. The second request to y costs again one and the element will be moved to the front.

For B, again the potential request to x is free and preserves the invariant. The first request to y costs 1 and leaves the list unchanged. Then the second request to y costs 1 again and moves y to the front. Each subsequent request for y or x in $(xy)^* y$ costs one and swaps the list, in the end the last request for y costs nothing, the list state is $[y, x]$ and the internal state (being the history) contains the prefix $[y, y]$. In total let k be the number of repetitions of the star expression, then the cost for B is $2 * k + 1$. We can express $k = (\lvert\sigma\rvert - 5)\ div\ 2$ and thus the cost to be $2 * (\lvert\sigma\rvert - 3)\ div\ 2$.

A similar derivation gives a total cost of $2 * (|\sigma| - 3)$ *div* 2 for type C.

The last type is easy, any request to x costs nothing, and the invariant is preserved.

The formal proof for each type is again as for the analysis of OPT2 a simulation of the algorithm plus an induction on the star of the regular expressions. The complicated part is to carry through the invariant involving the history of the request sequence.

Note that in contrast to Table 1.1 in [9, section 1.6.1] we are not able to state the respective costs relative to the number of repetitions of yx, nevertheless it can be stated in terms of the length of the request sequence. Albeit following their idea, we define the phase types differently: They allow more than one heading x in type A, B and C; in contrast we only allow zero or one occurrence but add type D. This captures the idea of splitting the request sequence into phases, that end with two consecutive requests to the same element, more precisely. This explains the differences of the results.

Finally, as the 4 types cover all request sequences of a phase (Lemma 7.9) we obtain the following:

Lemma 7.11. *For any request sequence qs in $x^? yy + x^? yx(yx)^* yy + x^? yx(yx)^* x + xx$ and the history h satisfying the invariant it holds:*
$T^*_{TS}\ [x, y]\ h\ qs \leq 2 * T^*\ [x, y]\ qs\ (OPT2\ qs\ [x, y]) \wedge$
$(\exists\, x'\, y'.$
$\quad s\text{-}TS\ [x, y]\ h\ qs\ |qs| = [x', y'] \wedge (\exists\, hs.\ rev\ qs\ @\ h = [x', x']\ @\ hs))$

7.2.3 Phase Partitioning

With the competitiveness results for the phases at hand we now turn to the whole request sequence:

Lemma 7.12. $T^*_{TS}\ [x, y]\ h\ \sigma \leq 2 * T^*\ [x, y]\ \sigma\ (OPT2\ \sigma\ [x, y]) + 2$

Proof. Suppose the initial list is $[x, y]$, the current history is h and the invariant for TS holds.

As we already mentioned we partition the given request sequence σ into phases.

We proceed by well-founded induction on the length of σ and chop off the phases one by one.

Thus we have two cases: either we have a phase as prefix of σ or σ is an incomplete phase (c.f. Figure 7.10).

In the first case it is possible to find a prefix xs that ends in two consecutive requests to the same element. Let ys denote the rest s.t. $\sigma = xs\ @\ ys$. Let LTS' denote the list state of TS after serving xs.

$$\begin{aligned}
& T^*_{TS}\ [x, y]\ h\ \sigma \\
= \ & T^*_{TS}\ [x, y]\ h\ (xs\ @\ ys) \\
= \ & T^*_{TS}\ [x, y]\ h\ xs + T^*_{TS}\ LTS'\ (rev\ xs\ @\ h)\ ys \\
\leq \ & T^*_{TS}\ [x, y]\ h\ xs + 2 * T^*\ LTS'\ ys\ (OPT2\ ys\ LTS') + 2 \\
\leq \ & 2 * T^*\ [x, y]\ xs\ (OPT2\ xs\ [x, y]) + 2 * T^*\ LTS'\ ys\ (OPT2\ ys\ LTS') + 2 \\
= \ & 2 * T^*\ [x, y]\ \sigma\ (OPT2\ \sigma\ [x, y]) + 2
\end{aligned}$$

We first split the serving of σ by TS at the end of the phase. Knowing from the second part of Lemma 7.11 that after serving the phase the invariant holds again, we can apply the induction hypothesis on the shorter request sequence *ys*. Then we use the first part of Lemma 7.11. After that we have to put together the serving of σ by OPT2. This is valid as we can show that OPT2 has the same list state (*LTS'*) as TS after serving the prefix *xs*: after the two consecutive requests to the same element this element is in front of OPT2's list, as it is for TS's.

In the second case if it is not possible to find a phase as prefix, we either have the trivial case of an empty request sequence or a request sequence with alternating x and y ($(x+1)(yx)^*y + (y+1)(xy)^*x$). But we can complete σ to a proper phase by repeating the last requested element: e.g. *xyx* gets padded to *xyxx* which is a valid type C phase.

$$
\begin{aligned}
& T^*_{TS}\ [x,y]\ h\ \sigma \\
\leq\ & T^*_{TS}\ [x,y]\ h\ (pad\ \sigma\ x\ y) \\
\leq\ & 2 * T^*\ [x,y]\ (pad\ \sigma\ x\ y)\ (OPT2\ (pad\ \sigma\ x\ y)\ [x,y]) \\
\leq\ & 2 * T^*\ [x,y]\ \sigma\ (OPT2\ \sigma\ [x,y]) + 2
\end{aligned}
$$

First we know that serving one more request costs at least as much as serving σ. Then we can apply lemma 7.11 as before. Finally we know that OPT2 has at most cost 1 for the padded request, which gets doubled and constitutes the constant term.

□

Now as we know that OPT2 is optimal and we can reintegrate our formulation of TS into the framework we obtain the corollary:

Corollary 7.13. $[\![x \neq y;\ set\ qs \subseteq \{x,y\}]\!] \Longrightarrow T^*_{on}\ rTS\ qs\ [x,y] \leq 2 * T^*_{opt}\ [x,y]\ qs + 2$

section "TS is 2-competitive"

Assuming that we also show that TS has the pairwise property we can use the list factoring lemma 5.9 and conclude that TS is 2-competitive:

Lemma 7.14 ([9, Theorem 1.4])**.**
$pairwise\ rTS \Longrightarrow compet^*\ rTS\ 2\ \{init \mid distinct\ init\}$

8 Open Questions

This chapter summarizes what has been formalized in this thesis, what is still to be proven and which results would be the next logical step to formalize. Furthermore open research problems about the list update problem as well as connection to other interesting areas are presented.

8.1 Open to formalize

fill the proof gaps

	formalized	complete proof
MTF	✓*	✓*
lower bound (deterministic)	✓*	✓*
BIT	✓	✓
List Factoring	✓	
OPT2	✓	✓
TS	✓	(✓)
COMB		
lower bound (randomized)		

Table 8.1: Overview of what has been formalized and proven in this thesis.

Table 8.1 shows what has been formalized and proven for the list update problem. Both MTF and the lower bound for deterministic algorithms have already been formalized by Nipkow [16] and were integrated into our framework. The lower bound is not described in this thesis. The analysis of BIT is verified in this thesis. Also OPT2 is formally proven to be optimal on lists of length 2. The list factoring technique is formalized but some proofs are still incomplete. For TS 2-competitiveness is proven, only the proof of pairwise property is left open. Proving the open goals would be an obvious direction for further work.

Formalize new results

Additionally, in [16] Nipkow formalized also the lower bounds for deterministic algorithms, its integration into the framework is a next step. Proving the lower bounds for randomized algorithm would indeed require way more work.

With the formalization of TS at hand, a similar development can be pursued for BIT: showing BIT has the pairwise property, and then analyzing the 4 types of phases for BIT. Some further work yields the result that COMB is 1.6-competitive.

There is a great variety of families of algorithms for the list update problem that would be interesting to study (including TIMESTAMP [2], SPLIT [13], COUNTER [21]).

Connection to Compression theory

An interesting application for the list update problem is compression. The open source compression framework bzip2 uses a combination of the famous Burrows-Wheeler Transformation, MTF and Huffman encodings [3]. In that regard any deterministic algorithm for the list update problem can be used to form a compression scheme. Details will be omitted here but c.f. [11] for more details. A thorough study of how the invertible Burrows-Wheeler Transformation manages to increase the locality of reference – which is then exploited by the list update algorithm – would be an interesting topic of study.

Analysis of other online problems

The framework for competitive analysis was designed to be applied to any online problem. Another direction of further work would be to analyse other online problems: such as Paging, online Graph Coloring, Bin Packing, etc.

8.2 Open Research Questions

The main open research questions in the field include the gap between the lower bound and the best competitive ratio for randomized algorithms of the list update problem.

It was long known that paid exchanges are necessary for the optimal offline algorithm. Recently it has been shown that paid exchanges not only give an advantage constant in the length of the request sequence but also a linear one [15]. Note that all online algorithms presented in this thesis and studied in literature only use free exchanges. An important question is how to integrate paid exchanges into online algorithms in a meaningful way; or how to show that they bring no further improvement. Furthermore it has been shown that online algorithms that satisfy the pairwise property have a lower bound of 1.6 [5] which is already attained by COMB. Consequently, to make progress in the gap considering algorithm without the pairwise property is necessary.

9 Conclusion

We formalized great parts of the first two chapters of Borodin and El-Yaniv's Book "Online computation and competitive analysis" [9]. This includes: a framework for the competitive analysis of randomized online algorithms, its interpretation for the list update problem, analysis of three popular algorithms (MTF, BIT and TS) and study of three techniques for their analysis (potential function method, list factoring and phase partitioning).

The formalization effort was reasonable, building upon the theory HOL-Probability and the already formalized analysis of MTF.

The framework could be used to analyse many other online algorithms, as well as the theory developed for the list update problem could be extended in various directions.

Bibliography

[1] S. Albers. A competitive analysis of the list update problem with lookahead. In *Mathematical Foundations of Computer Science 1994*, pages 199–210. Springer, 1994.

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

[3] S. Albers and S. Lauer. On list update with locality of reference. In *Automata, Languages and Programming*, pages 96–107. Springer, 2008.

[4] S. Albers, B. Von Stengel, and R. Werchner. A combined bit and timestamp algorithm for the list update problem. *Information Processing Letters*, 56(3):135–139, 1995.

[5] C. Ambühl, B. Gärtner, and B. Von Stengel. Optimal projective algorithms for the list update problem. In *Automata, Languages and Programming*, pages 305–316. Springer, 2000.

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

[7] R. Bachrach, R. El-Yaniv, and M. Reinstadtler. On the competitive theory and practice of online list accessing algorithms. *Algorithmica*, 32(2):201–245, 2002.

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

[9] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. cambridge university press, 2005.

[10] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

[11] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[12] J. Hölzl, A. Lochbihler, and D. Traytel. A formalized hierarchy of probabilistic system types. In *Interactive Theorem Proving*, pages 203–220. Springer, 2015.

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

[14] S. Kamali and A. López-Ortiz. A survey of algorithms and models for list update. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 251–266. Springer, 2013.

[15] A. López-Ortiz, M. P. Renault, and A. Rosén. Paid exchanges are worth the price. In *32nd International Symposium on Theoretical Aspects of Computer Science*, page 636, 2015.

[16] T. Nipkow. Amortized complexity verified. *Archive of Formal Proofs*, July 2014. http://afp.sf.net/entries/Amortized_Complexity.shtml, Formal proof development.

[17] T. Nipkow. Amortized complexity verified. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP 2015)*, volume 9236, pages ?–?, 2015.

[18] T. Nipkow and D. Traytel. Unified decision procedures for regular expression equivalence. *Archive of Formal Proofs*, Jan. 2014. http://afp.sf.net/entries/Regex_Equivalence.shtml, Formal proof development.

[19] N. Reingold and J. Westbrook. *Randomized algorithms for the list update problem*. Yale University, Department of Computer Science, 1990.

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

[21] N. Reingold, J. Westbrook, and D. D. Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994.

[22] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, 1976.

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