Improvements to Online Learning Algorithms with Applications to Binary Search Trees

by Jussi Kujala

Contact Information:

	Jussi Kujala
mail:	Tampere University of Technology
	Department of Software Systems
	P.O.Box 553
	FIN-33101 Tampere
	Finland
tel:	+358 3 3115 2700 (office)
	+358 40 5686 235 (mobile)
fax:	+358 3 3115 2913
e-mail:	jussi.kujala@tut.fi

Abstract

In this work we are motivated by the question: "How to automatically adapt to, or learn, structure in the past inputs of an algorithm?". This question might arise from the need to decrease the amount of consumed resources, such as run-time or money. In addition, we also consider, in part, the growing significance of the I/O-issues in computation. A major focus is on studying data structures, such as binary search trees (BSTs). They are a common and practical solution to the problem of representing data such as sets; hence it is important to understand their properties.

For the most part we work with algorithms which continuously serve requests under uncertainty about future inputs. These algorithms are called online algorithms. To analyze their performance we use the competitive analysis. In it the uncertainty over future inputs is not necessarily modeled stochastically, but rather we deal with the uncertainty by comparing our own performance to how well we could have done had we known the input sequence.

In the first part of the Thesis we study a general online setting, where one repeatedly chooses a decision from a fixed number of options and then observes a bounded cost for that decision. We consider using a known Follow the Perturbed Leader algorithm in this setting. Previously it has been successfully applied in a related online setting, where we know costs on all past decisions. As a new result we find that similar algorithms also perform well when applied in the more restricted setting we study.

In the second part of our Thesis we study BSTs. We first concentrate on BST algorithms that may change the structure of the BST during the

accesses. We give a lower bound to the cost of any BST algorithm in terms of the entropy of the source generating the accesses and also in terms of the complexity of the access sequence. As an application to these bounds we show that the splay tree is optimal on accesses that are generated from a Markov chain with a spatial locality of reference.

Constructing an actual online BST algorithm, which is competitive versus any full-information BST algorithm on every access sequence, has recently attracted attention. We give new insights on how to implement an almost competitive online data structure. Our approach is based on augmenting a standard balanced structure, such as red-black tree or B-tree, with dynamic pointers that cache directions of the previous accesses. We can easily modify the overhead incurred by this scheme by changing the number of dynamic pointers.

Finally, we give a simple algorithm that computes an approximately optimal static BST for given weights on items. Although this is a classic problem with several previous solutions, we are the first to give an I/O efficient approach that produces a well-performing output.

Preface

> An expert is a person who has made all the mistakes that can be made on a very narrow field –Niels Bohr

Even after writing this Thesis I think I have a long way to go before I am an expert, but I have taken the first steps. I have enjoyed walking these steps, and perhaps this is because of the challenge and freedom in the academic world, together with the purpose of hopefully creating something new and useful. My method of navigation has primarily been the Brownian motion, which can take you to unexpected places, but unfortunately not so far as traveling on a straight line.

This preface is the place to acknowledge the contribution of other people, for without them this Thesis would not be. I appreciate the fact that many people have contributed to my research work, and also in ways that are not acknowledged. Especially, I have found comments from different viewpoints very insightful. Though the articles and books cited in this Thesis are an important source of information I learned during my Ph.D. studies, I also had other valuable sources not typically cited, such as Wikipedia (cough), blogs, and lecture slides. Thank you all who make these sources available!

The work presented in this Thesis was carried out at the Department of Software Systems in the Tampere University of Technology during the years 2004-2007; I thank my coworkers and especially my advisor, Tapio Elomaa, for the interesting discussions and help they have provided to me. I also thank those people who gave comments on this introduction to the

attached publications: Timo Aho, Tapio Elomaa, Mikko Leppänen, Tommi Mikkonen, and Matti Saarela.

I was financially supported by Tampere Graduate School in Information Science and Engineering (TISE), and I gratefully acknowledge them the three years of generous funding they gave. I also acknowledge the funding provided by Academy of Finland (through projects) and the Nokia Foundation (through scholarships).

Finally, I thank my parents, Anne and Antero, for their support and case.

Tampere, 7th September 2008
Jussi Kujala

Contents

Abstract i

Preface iii

List of Publications vii

List of Abbreviations xiii

List of Symbols xv

1 Introduction 1
- 1.1 Background 1
- 1.2 The Structure of the Thesis 3
- 1.3 Summary of the Main Contributions 6
- 1.4 Author's Contribution 7

2 Preliminaries 9
- 2.1 Online Algorithms and Setting 9
- 2.2 Competitive Analysis 11
- 2.3 The Expert Setting 13
- 2.4 Online Geometric Setting 15
- 2.5 Follow the Perturbed Leader 16

3 Bandit FPL 19
- 3.1 The Bandit Setting 19

3.2 Non-Adaptive Adversary . . . 21
3.3 Adaptive Adversary . . . 23

4 On the Competitive Analysis of BSTs 25
4.1 Background on the BSTs . . . 25
4.2 Counting Costs in BSTs . . . 26
4.3 Efficient Use of FPL with BSTs . . . 27
4.4 Approximate BSTs Work with FPL . . . 30
4.5 Offline BST Algorithms . . . 36
4.6 Limits of BST algorithms . . . 38
4.7 Augmenting BSTs to Become Dynamic . . . 41
4.7.1 Wilber's lower bound for BST algorithms . . . 42
4.7.2 Tango: the almost competitive BST algorithm . . . 43
4.7.3 Our approach for a competitive BST algorithm . . . 43
4.7.4 Discussion on the Poketree . . . 44

5 Computing I/O Efficiently an Approximate BST 47
5.1 Background . . . 47
5.2 Costs in Data Structures . . . 48
5.3 Our Algorithm: AWOBST . . . 50
5.3.1 Description of AWOBST . . . 51
5.3.2 An example . . . 52
5.4 I/O performance of the constructed BST . . . 54
5.4.1 Where to place the items? . . . 54
5.4.2 How to place the items? . . . 55

6 Conclusions 59

List of Publications

This Thesis is a compound of the following publications and an introduction to them. In the text, these publications are referred to as [P1] - [P5].

[P1] Jussi Kujala and Tapio Elomaa, “On Following the Perturbed Leader in the Bandit Setting” in *Proceedings of the 16th International Conference of Algorithmic Learning Theory*, Lecture Notes in Computer Science, volume 3734, p. 371–385, 2005.

[P2] Jussi Kujala and Tapio Elomaa, “Following the Perturbed Leader to Gamble at Multi-Armed Bandits”, in *Proceedings of the 18th International Conference of Algorithmic Learning Theory*, Lecture Notes in Computer Science, volume 4754, p. 158–172, 2007.

[P3] Jussi Kujala and Tapio Elomaa, “The Cost of Offline Binary Search Tree Algorithms and the Complexity of the Request Sequence”. *Theoretical Computer Science*, volume 393 issue 1-3, p. 231-239, 2008.

[P4] Jussi Kujala and Tapio Elomaa, “Poketree: A Dynamically Competitive Data Structure with Good Worst-Case Performance” in *Proceedings of the 17th International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science, volume 4317, p. 277–288, 2006.

[P5] Jussi Kujala, “Assembling Approximately Optimal Binary Search Trees Efficiently Using Arithmetics”, Submitted for publication, 2007.

List of Figures

1.1 A conceptual picture on online algorithms. 2

4.1 An illustration of a part of the decision set $\mathbb{S}^2$ 32
4.2 An illustration of how a P-tree changes during an access. . 42
4.3 An example of a Poketree data structure. 45

5.1 The BST that AWOBST constructs from the input given in Table 5.2. 53
5.2 How a BST is embedded to a larger tree. 56

List of Tables

2.1 Follow the Perturbed Leader (FPL) algorithm. 18

4.1 The online BST-FPL algorithm. 28
4.2 Requirements for OPT-BST algorithm. 29
4.3 Definitions of different types of optimalities for BST algorithms. 37
4.4 Known asymptotic upper bounds on the performance of the competitive BST algorithms. 46

5.1 Performance of selected algorithms for constructing weighted BSTs. 51
5.2 An example of an input to AWOBST. 52
5.3 Cumulative probabilities and the priorities of the items in AWOBST, calculated from the input in Table 5.2. 53
5.4 How to I/O efficiently place the nodes to the memory, a general approach. 57

List of Abbreviations

AWOBST	Arithmetic Weight-Optimal Binary Search Tree
B-FPL	Bandit Follow the Perturbed Leader
BST	Binary Search Tree
BST-FPL	Binary Search Tree Follow the Perturbed Leader
FPL	Follow the Perturbed Leader

List of Symbols

x	A general unknown variable that depends on the context.
σ	An access sequence consisting of items from σ_1 to σ_T.
T	The length of an access sequence (often denoted by σ), or a total number of time turns an online algorithm executes.
H_a	A heap with an index a.
$\vec{p}$	A probability distribution with probabilities $\langle p_1, \ldots, p_i, \ldots, p_n \rangle$ on items $\{1, \ldots, n\}$.
$H(\vec{p})$	Entropy of a draw from distribution $\vec{p}$.
$\mathbf{cost}(x)$	Cost of the algorithm x.
APP	An online algorithm.
OPT	The optimal algorithm or an oracle returning optimal decisions.
$\mathbb{S}^n$	A subset of $\mathbb{R}^n$ defined on page 32.
B	The length of a cache line or a page size.
S_D	The set of decision vectors in FPL.
M_D	The diameter of the set S_D.

S_C	The set of cost vectors in FPL.
M_C	The diameter of the set S_C.
M	The maximum cost during one turn in FPL, also the maximum of dot products between a vector from S_D and a vector from S_C.
$\vec{c}$	A cost vector.
$\vec{c}_t$	A cost vector during the time turn t.
$[\![\vec{c}]\!]_i$	The ith coordinate of a cost vector.
$\hat{\vec{c}}$	A randomized estimate of a cost vector.

Chapter 1

Introduction

> Computing is fundamentally about solving problems using strictly defined processes under resource constraints –Derrick Coetzee

1.1 Background

Algorithms describe precisely the process of computing, and so to effectively solve problems we must understand them. A key property of an algorithm is the performance: what is the cost of using an algorithm? Unfortunately it may be that predicting the performance of an algorithm is difficult. As examples consider the simulated annealing and simplex algorithms in optimization, quicksort for sorting, and algorithms that handle caching. In these examples the good performance was originally observed empirically, and more theoretically sound explanations have followed (if have) the observations. A complicating factor in these examples is that a simple worst-case analysis of resources does not work. Naturally we want to know what makes algorithms work well, so that we can exploit this knowledge when designing algorithms.

This Thesis primarily concerns a class of algorithms called online algorithms which, in general, are somewhat difficult to analyze. Unfortunately the term "online algorithms" is ambiguous and it has several meanings de-

pending on the context. In this text the term online refers to how the input is given: the algorithm continuously serves requests and the cost of completing a request may depend on how the past requests were served; see the rough sketch in Figure 1.1. Some examples that fit well to this setting

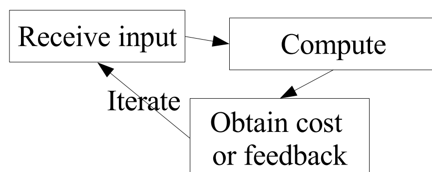

Figure 1.1: A conceptual picture on online algorithms.

are network routing, prediction of stock prices, and page caching, to list a few. Other problems, such as data structures, are not as natural examples, because they have been analyzed successfully with worst-case costs. Nevertheless, they may also benefit from the tools used in the online setting to analyze behavior arising from structure in the input.

The title of this Thesis refers to "learning", because in online problems there is uncertainty about future inputs and basically the way to perform well is to adapt to past input and hence learn. When we face uncertainty then our method of choice is often to model the uncertainty as stochastic, but our source of inputs does not necessarily conform to this choice. We use non-stochastic approach which relies on the competitive analysis (or the online adversial setting) to analyze how good an algorithm is [19]. In short, the competitive analysis measures performance by comparing how well an online algorithm performs against some set of offline algorithms or strategies that are aware of the future inputs. So, in the competitive analysis we prove statements such as:

> Our algorithm, for any possible sequence of inputs, is never worse than two times the cost of the best algorithm in the set S for that sequence of inputs,

where S is the set of algorithms the online algorithm is competing against.

The cost of the algorithm depends, of course, on the application. For example, it could be the time used in computing, money lost, or the probability of wrong prediction by our online algorithm.

The competitive analysis is a tool that allows us to derive useful information on online algorithms, because other methods, such as measuring the absolute cost, may give little information on the suitability of the online algorithm. However, neither is the competitive analysis the silver bullet[1] for online problems. For example, no input is usually given a priority over the others, which may be a handicap in some problems, but nevertheless, competitive analysis provides a useful viewpoint on the algorithmic performance.

1.2 The Structure of the Thesis

We now summarize the content of the subsequent chapters, in which we introduce the attached publications and put our results to the wider context. In short, we have worked on problems where the motivation was at least originally related to data structures, like the traditional binary search trees.

First, Chapter 2 gives the preliminaries needed to understand the subsequent chapters, for example, it defines the online setting and the competitive analysis. In it we also recall the Follow the Perturbed Leader algorithm (FPL) for an online learning setting which generalizes many online decision problems of interest [47]. If we know the costs for all possible decisions in the past, then the FPL performs well versus the best static decision chosen in hindsight. For example, we can apply the FPL algorithm to online routing in a graph or serving accesses with a data structure. The objective is to perform as well as the best route or the best binary search tree chosen after the accesses have been revealed. The costs of the decisions are generated either by a non-adaptive opponent that is oblivious to our actions, or an adaptive opponent that is aware of our actions and can set the future

[1] The term "silver bullet" is a metaphor for a simple and effective solution for a problem for which there is arguably none.

costs according to this knowledge. When compared to other algorithms for similar problems FPL has advantages such as simplicity and flexibility.

Then Chapter 3 presents an introduction to Publication [P1]. It shows that if we restrict the information about the past costs in FPL, namely that we get to know only the cost of the decisions which we made, we can also use an approach similar to the FPL to perform well. This restriction that we receive only partial information is relevant in several problems, for example when we make series of runs with an option of selecting from several different algorithms and we want to perform as well as the best among them. The performance bound is asymptotically as good as what is known for the previous EXP3 algorithm against a non-adaptive adversary [10]. The algorithm itself is somewhat more general as it is not bounded by uniform bounds for the costs of the experts. For problems where the adversary can adapt to our own actions Chapter 3 gives a variant that also achieves a regret bound matching that of previous work. This variant is studied in detail in Publication [P2].

Chapter 4 proceeds to competitive analysis of binary search trees (BSTs). Essentially the aim of this analysis is to study if, when, and how should we change the structure of the BST that is used to serve accesses. Section 4.6 introduces Publication [P3], which lower bounds the cost of a BST algorithm serving accesses that are generated from a random source (note that the BST that the algorithm uses may change, at a cost, between the accesses). Additionally, it shows that for a random source, which gives a probability $\mathbf{P}(\sigma)$ to a sequence of accesses σ, the cost of any binary search tree algorithm is lower bounded by a constant times entropy that is calculated from these probabilities $\mathbf{P}(\sigma)$ on sequences. Previous work has considered only the case in which the individual accesses were generated independently and identically, and our result generalizes to any stochastic source. The result implies that splay trees are asymptotically as good as any binary search tree algorithm on an access sequence that is generated from a Markov source with a *spatial* locality of reference. The spatial locality of reference means that the next access is more likely to be generated from items near the previously accessed item. Also, the cost of serving an access sequence σ is bounded with the complexity of the sequence σ as

measured by the Kolmogorov complexity of σ.

Section 4.7 introduces Publication [P4], which considers online binary search tree algorithms that perform well against the best binary search tree algorithm. We find that we can implement the approach of Demaine et al. [32] — which is $\mathcal{O}(\lg\lg n)$-competitive versus any binary search tree algorithm — on top of a traditional balanced search structure, such as the red-black tree, hence inheriting the good qualities of these structures, like a good worst-case cost. The resulting structure also supports updates more naturally than the previous $\mathcal{O}(\lg\lg n)$-competitive data structures and we can easily regulate the overhead incurred by this approach.

Chapter 5 continues to discuss Publication [P5] which concerns a slightly different, but related, topic. It gives an algorithm to construct an approximately optimal BST when we have known weights on the items. The goal is to minimize the expected cost of an access for these weights. This topic is well researched and the difference to the previous work is that our construction is I/O-efficient, i.e., for n items and a cache with lines of size B the cost of our approach is $\mathcal{O}(n/B)$ in the cache-oblivious model. Also, the cost is $\mathcal{O}(n)$ in the standard unit cost model. The motivation is that the relative importance of I/O has grown as the environment where the algorithms execute has underwent change.

Surprisingly, we also obtain a bound on the performance of the resulting BST that is slightly better than previously. The previous best bound guaranteed that the item is on average found after searching

$$H + 1$$

nodes of the BST, where the entropy H is computed from the probabilities p_i on the items: $H = \sum_{i=1}^{n} -p_i \lg p_i$. For the optimal BST we give an upper bound

$$H + \lg(1 + p_{\max}) + 0.087,$$

where $p_{\max}$ is the maximal probability on the items. Also, for the special case when $p_{\max} < 2/3$ we give an upper bound $H + 0.503$.

This topic is connected to the rest of the Thesis in Section 4.3, where we first recall how we can apply FPL algorithm to BSTs when we have access

to an algorithm computing optimal BST for given weights. The resulting algorithm performs almost as well as the best BST in hindsight for any access sequence, but it needs to occasionally compute the optimal BST. We observe that to formally guarantee good performance under the competitive analysis versus the best static BST it is enough to compute an approximate BST.

1.3 Summary of the Main Contributions

To summarize, the main contributions of this Thesis are:

1. In [P1,P2] we show that we can apply the FPL algorithm in the bandit setting, against both non-adaptive and adaptive opponent.

2. In Section 4.4 we observe that using an approximation algorithm with the FPL algorithm in the application of BSTs results in a good performance.

3. In [P3] we give a lower bound to the cost of any BST algorithm, which is constant times entropy of a probability distribution over access sequences. Also, the cost is at least a constant times the Kolmogorov complexity of the access sequence.

4. In [P4] we demonstrate how to make a $\mathcal{O}(\lg \lg n)$-competitive datastructure on top of a traditional balanced search structure.

5. In [P5] we give an I/O-efficient algorithm for computing an approximately optimal BST when we have weights on the items. We also give a upper bound to the performance of this BST which is slightly better than previously known. In addition, in Section 5.4 we observe that we can place the constructed BST in a way that the accesses to it are served I/O efficiently.

1.4 Author's Contribution

In Publications [P1,P2,P3,P4] the author came up with the idea after given a general direction and carried out the research work and significant part of the writing. The author is the sole author of Publication [P5].

Chapter 2

Preliminaries

This chapter gives a short introduction to the basic concepts that we use in the following chapters 3 and 4. These are the online setting, competitive analysis of its algorithms, the more specific expert and geometric settings, and the Follow the Perturbed Leader (FPL) algorithm.

2.1 Online Algorithms and Setting

In problems of the *online setting* we serve a sequence of inputs without knowledge of the future inputs. An online algorithm solves the problem indicated by the input, obtains feedback or cost, and then waits for the next input. Let us illustrate an online problem with a simple and well-known example.

Example: the paging problem. *Programs store information in a finite memory. The memory may naturally fill up and then one solution is to temporarily evict a part of the information to a larger and slower media. Information is stored as units called pages, and the problem of choosing a page in the memory to evict is called the* paging problem*. In this problem the sequence of inputs consists of the accesses to the memory and the cost of an algorithm is the time waited for the accesses to complete. This problem is also called the* caching problem*, where the finite memory is referred to as*

a cache which contains cache lines that correspond to pages. The situation in which a page is referenced, but it is not in the fast memory, is called a page fault, or alternatively a cache miss.

In the above problem the cost is a function of both the sequence of accesses and the sequence of eviction decisions made by the online paging algorithm. Worth noting is that the decisions do not matter independently in the cost function, i.e., the cost is not a simple sum over the eviction decisions, because the contents of the cache affects the cost. Hence, an algorithm for the page caching has to consider serving future inputs, which naturally are uncertain. We can consider this as attempting to use information we collect in the past to predict the future. There is a vast amount of literature on online algorithms for different online problems; for a more comprehensive introduction see for example [19, 16, 5, 3, 18].

If we want to design a good algorithm we must, of course, know how good it is. We first point out the deficiencies that the simple worst-case cost has in some online problems.

The problem is that the worst-case cost does not necessarily give any information on the performance of an algorithm in an online problem. For example, in the page caching problem the worst-case scenario happens if we have a program that inspects the contents of the cache and makes a reference to a page that is not in the cache. In this case all algorithms for page caching suffer the same worst-possible cost.

Note that we generated this worst-case behavior through an actual algorithm that makes memory accesses, and the worst-case was not counted over individual fixed sequences of accesses. This is not an important distinction, because the average cost (and hence the worst-case cost) over all sequences is also uninformative. On the average case all pages are equally probable for the next access and then all algorithms must witness, on the average, the same cost, which is approximately

$$1 - \frac{\text{number of pages in the cache}}{\text{the total number of pages}}$$

page faults per memory access.

The example we used was an extreme one. In data structures, for example, the worst-case analysis has been very successful. However, the worst-case analysis alone does not necessarily characterize typical performance on data structures, as another well-known example shows.

Example: The Move to Front Rule [64]. *We store data in a simple linked list, and when an access arrives then we scan for the accessed item and move it to the front of the list. This is called the Move to Front (*MTF*) rule. In some situations it empirically works quite well, even if the worst-case cost is $\mathcal{O}(n)$ for each access [15].*

Now the questions arise how good the MTF-rule is, and when to use it (or is there something better?). Naturally it performs well in cases such as when only one item is accessed, but for random accesses it is as bad as any list. In the next section we recall the *competitive analysis* which is an analysis tool that gives insight to the performance of the MTF-rule and other online algorithms.

2.2 Competitive Analysis

The *competitive analysis* or the *adversarial setting* is a method to measure the performance of online algorithms, which were defined in the previous section. In it the cost is not directly measured, but rather the cost is compared to the cost of other algorithms or strategies for the same problem. For example we could compare an online algorithm to the best possible *offline* algorithm which knows the future inputs and is thus the full-information version of the same problem.

If APP is an online algorithm and OPT the corresponding offline algorithm, then over all input sequences σ the competitive analysis calls for bounds of the form

$$\mathbf{cost}(\text{APP}\,(\sigma)) \leq c_{\text{mul}}\,\mathbf{cost}(\text{OPT}\,(\sigma)) + c_{\text{add}},$$

where c_{mul} and c_{add} are constants that characterize how good the bound is. The constants may depend on the size of the problem instance, such

as size of the cache in the page-caching, but this bound must hold over all inputs σ. If in the problem at hand the online algorithms are a subset of the offline strategies then naturally we have a lower bound

$$\mathbf{cost}(\text{APP}\,(\sigma)) \geq \mathbf{cost}(\text{OPT}\,(\sigma))\,.$$

Thus, if our bound for an online algorithm is good against an offline algorithm, then we at least know that we cannot perform much better. Unfortunately, in some problems online and offline algorithms are strictly separated in their performance. In these problems the competitive analysis can not always guarantee good performance in practice, because the inputs that we encounter can often be far from the worst-case ones.

As an example, in the paging problem, the Least Recently Used (LRU) algorithm that evicts the page that was used least recently has the following bound [64]

$$\mathbf{cost}(\text{LRU}) \leq k\,\mathbf{cost}(\text{OPT})\,,$$

where k is the size of the cache and OPT is the optimal algorithm that knows all future memory accesses and evicts the page according to this information [14]. This bound is already better than nothing, although the multiplicative term k is high (note that no deterministic algorithm can do better against the OPT). Hence this bound gives useful information on the performance of LRU, in contrast to the worst-case and the average case discussed earlier.

As another example, for the MTF rule discussed at the end of the previous section, the following result is known [15].

$$\mathbf{cost}(\text{MTF rule}) \leq 2\,\mathbf{cost}(\text{best static list})\,.$$

In this Thesis we focus solely on the competitive analysis to analyze online algorithms and we do not consider alternatives. However, these are important because it is known that the performance of an online algorithm is not necessarily characterized completely by the competitive analysis [35]. A list of alternative methods is, for example, given in [34] or in [8], where the authors also formalize and analyze the intuitive notion that LRU optimally captures the temporal locality in the accesses.

2.3 The Expert Setting

The expert setting of online learning is a general framework, where we have n experts that give black-box advice on what to do [52, 71, 38]. We get to know how good their advice was after acting according to an expert of our choice. We must act on a number of time turns and we choose an expert during each of them. For example, the experts could give advice on the length or temperature of the winter, or predict the next bit in a data stream [23], or assess the performance of different paging policies [17].

Our decision of an expert to select is affected by how we believe the costs are assigned. Below we discuss three possible ways to model these costs, ordered from the weakest to the most powerful.

1. In *stochastic* generation of the costs we model them with a probability distribution, which for example could generate costs on different time turns independently and/or identically, or from a Markov chain, or from another relevant distribution. In this Thesis we do not consider the stochastic generation.

2. In *non-adaptive* or *oblivious* generation we recognize that not all inputs come from a source that is (approximately) stochastic. Hence, the approach is as general as possible; we assume that the input can be anything, except that our own actions do not affect the future costs. Also, we have a uniform or expected upper bound for the cost that an individual expert can suffer during one time turn. This implies that we can think that we generate the sequence of costs, for each of the n experts, and for each of the possible time turns, before the online algorithm executes.

3. In *adaptive* or *non-oblivious* generation we take away the restriction that our own actions do not affect the future costs. Hence, the generated costs may depend on our past actions. For example, if the experts represent different paths and we must route traffic through one path, then the cost of a path can change depending on how much traffic we have routed through it in the past. In theory, our actions may often

affect future costs through hypothetical feedback effects, though the effect is in reality negligible. For example, in the paging problem our choices can change the future memory references by evicting always a page belonging to a particular process. This effectively breaks the usual non-adaptive analysis.

Of course, these cost assignments may or may not be realistic depending on the particular application. Therefore, it is important to assess what assumptions are reasonable in the application under consideration. For example, Awerbuch et al. [11] argue that a pessimistic adversial model may be relevant in routing as a network might be attacked by a malicious and intelligent outsider. However, if the faults in the routing are random, then a greedy approach is feasible. Also, there recently was a workshop dedicated to problems where the adaptive nature of the opponent is relevant [59]. These problems were related to security issues, where the opponent is malicious and intelligent.

We define the expert setting more formally as follows. We have n experts and for each time turn $t \in \{1, \ldots, T\}$ we have a cost vector $\vec{c}_t$, where the coordinate $[\![\vec{c}_t]\!]_i \in [0, 1]$ is associated with the cost of the expert $i \in \{1, \ldots, n\}$. On turn t we select an expert $\bar{i}(t)$ and we would like to minimize our (expected) total cost

$$\mathbf{cost}(\textsc{app}) = \mathbf{E}\left(\sum_{t=1}^{T} [\![\vec{c}_t]\!]_{\bar{i}(t)}\right),$$

where the expectation is over our own actions. Bounding the above cost is infeasible, so we use the competitive analysis and compare the above absolute cost to the cost of the best expert in hindsight

$$\mathbf{cost}(\textsc{opt}) = \min_i \sum_{t=1}^{T} [\![\vec{c}_t]\!]_i .$$

Hence, we minimize our *regret*

$$\mathbf{regret}(\textsc{app}) = \mathbf{cost}(\textsc{app}) - \mathbf{cost}(\textsc{opt}) .$$

In the expert setting our aim is to prove rigorous guarantees for the regret instead of using ad-hoc heuristics. So, we perform poorly if and only if every expert is incompetent.

Note that a rigorous bound in terms of the absolute cost was infeasible, because with a non-adaptive or adaptive opponent it is possible that on each turn all but one random expert incurs costs. Then we cannot bound the difference between APP and the offline algorithm which selects the best individual expert each turn. Note also that in the expert setting the constant factors in the regret bound are more important than usually with algorithms. The rationale is that the regret can measure, for example, money, which is more important than the execution time of an algorithm, which is often relatively cheap and also difficult to measure with regards to constant factors due to differences between platforms and implementation decisions.

Next we review a generalization of the expert setting that is useful in certain problems.

2.4 Online Geometric Setting

The online geometric setting was introduced in [47]. In it we have a set of decisions S_D in a vector space $\mathbb{R}^d$, and we associate each vector with an expert or a possible decision that we can choose. On turn t the cost vector $\vec{c}_t \in S_C \subset \mathbb{R}^d$ gives the cost of choosing any decision $\vec{d} \in S_D$ as the Euclidean dot product $\vec{d} \cdot \vec{c}_t$. For example, the original expert setting is obtained by choosing the decision vectors as the standard basis vectors $e_i = (0, \ldots, 0, \underbrace{1}_{i\text{th}}, 0, \ldots, 0)$ and listing the costs of the experts in the cost vector $\vec{c}$.

The performance of the algorithms in the online geometric setting does not depend on the number of possible decisions in the comparison class, which is the usual case for the algorithms of the expert setting. Rather, the performance depends on the diameters of the sets S_D and S_C, which makes this setting useful, as the following example shows.

Example: Selecting a BST. *Associate a binary search tree with d items to a d -dimensional vector by listing the depth of each item in the binary search tree. The cost vector $\vec{c}$ of the online geometric setting is an indicator vector for an access, and hence the dot product of $\vec{c}$ and the decision vector equals the cost (defined as the depth of the accessed item). The diameter of the decision set S_D is d^2 and the diameter of the cost set S_C is 2. These are lower than the number of decision vectors $\binom{2d}{d}/(d+1)$ which is approximately 4^d using Stirling's approximation.*

Other problems that fit to the online geometric setting are the linked list data structure structure, *online routing* [47], *online set cover* [46], *online traveling salesman problem* [46], and the setting is also useful in more general online convex optimization [76].

2.5 Follow the Perturbed Leader

There are several algorithms for the expert setting, and we do not introduce this vast literature which is reviewed for example in the book [24]. Let us, however, introduce the FPL algorithm, because part of our work is related to it.

FPL was originally proposed by Hannan [42] and reintroduced by Kalai and Vempala [47]. It is the only algorithm for the online geometric setting we know of. The FPL algorithm has an intuitive interpretation. When we seek good performance in the expert setting then an obvious thing to try is to choose the best expert for the costs observed so far, i.e., follow the leader. Unfortunately, every deterministic algorithm fails under the competitive analysis, because of the following strategy for the adversary. On each turn set the cost of the expert that will be chosen by the deterministic algorithm to one and give a cost of zero to all others. Hence, during T turns the deterministic algorithm incurs a cost of T, but because of the pigeon-hole principle (or rather the reverse of it) there is an expert with a cumulative cost of at most T/n. Thus, the regret is not bounded with anything useful.

This is unfortunate, however FPL resembles the idea of following the leader. In FPL the leader is chosen after perturbing the costs with random-

ness, e.g. after adding a random vector to the perceived cumulative cost vector. The actual algorithm is given in Table 2.1, where we also list the conditions for the random perturbation distribution. Note that in FPL we must be able to select the best decision given the past costs and in the online geometric setting it is problem specific whether we can do this.

The performance of FPL depends on the structure of the sets S_C and S_D and not on the number of decisions. For example for a uniform perturbation distribution the competitive bound for FPL is [47]:

$$\mathbf{E}(\mathbf{cost}(\textsc{fpl})) \leq (\text{cost of the best decision}) + \epsilon\, M\, M_C\, T + \frac{M_D}{\epsilon},$$

where M is a maximum cost during one turn and M_C and M_D are the diameters of the cost set S_C and the decision set S_D. The regret is $2\sqrt{M\, M_C\, M_D\, T}$ for a properly chosen value of ϵ.

This bound can not be compared to previous work as such, because we do not know of other algorithms for the online geometric setting. For the special case of the expert setting, we can compare the above regret bound to the regret bound $\mathcal{O}\big(\sqrt{T \lg n}\big)$ which is achieved by a variant of weighted majority algorithm [53] called Hedge [38]. The regret of the FPL with uniform perturbation is $\mathcal{O}\left(\sqrt{n\,T}\right)$. This demonstrates that for a uniform perturbation distribution the bound is not optimal in this problem. However, we can achieve a similar performance, if we choose the perturbation distribution to two-sided exponential distribution, or we can even achieve the same decisions as the Hedge algorithm, if we choose the perturbation carefully [30].

In addition to the advantage of flexible cost and decision sets, FPL has the advantage of so-called lazy behavior versus a non-adaptive opponent. This means that we do not have to select a new decision vector each time turn, but can stick to the same decision vector for approximately $\sqrt{T}$ time turns [47]. This is useful if we must pay to change the decision, for example if decisions are algorithms with internal state like a binary search tree.

We have the following bounds:

- for the diameter of the decision set: $\left\|\vec{d} - \vec{d'}\right\|_1 \leq M_D$ for all $\vec{d}, \vec{d'} \in S_D$,
- for the cost set: $\|\vec{c}\|_1 \leq M_C$ for any $\vec{c} \in S_C$, and
- for the maximum cost: $| \vec{c} \cdot \vec{d} | \leq M$.

Also, we have an oracle OPT that, for any cost vector $\vec{c}$, returns the best decision $\text{OPT}(\vec{c})$. The perturbation distribution $\mathcal{D}(\epsilon)$ with parameter ϵ is such that

- For a sample $\vec{\mu} \propto \mathcal{D}(\epsilon)$ the expected norm $\mathbf{E}(\|\vec{\mu}\|_\infty)$ is bounded with $\mathcal{O}(1/\epsilon)$ (ignoring the dependence on d).
- For any two cost vectors $\vec{c}_A$ and $\vec{c}_B$ and samples $\vec{\mu}_A$ and $\vec{\mu}_B$ from $\mathcal{D}(\epsilon)$ the expected difference $\mathbf{E}(\|\text{OPT}(\vec{c}_A + \vec{\mu}_A) - \text{OPT}(\vec{c}_B + \vec{\mu}_B)\|_1)$ is $\mathcal{O}(\epsilon \|\vec{c}_A - \vec{c}_B\|_1)$.

On time turn t the algorithm FPL does the following:

1. Chooses a random perturbation vector $\vec{\mu}_t \propto \mathcal{D}(\epsilon)$.
2. Picks the decision $\text{OPT}(\vec{c}_{1:t-1} + \vec{\mu}_t)$, where $\vec{c}_{1:t-1}$ is a shorthand for the cumulative cost vector $\sum_{\tau=1}^{t-1} \vec{c}_\tau$.

Table 2.1: Follow the Perturbed Leader (FPL) algorithm which is introduced in [47].

Chapter 3

Bandit FPL

The end of the previous chapter introduced the FPL algorithm for the expert setting. Publications [P1,P2] study how to apply the FPL in a more restricted setting, called the bandit setting. This chapter introduces these publications.

3.1 The Bandit Setting

The *bandit setting* is similar to the expert setting, but instead of getting knowledge on the performance of each expert on every turn, we obtain only the costs associated with the experts we choose to use. This situation is often depicted in terms of slot machines, where we have n slot machines and after playing one we, of course, only know how much we profited or lost with that particular machine. This setting is also called the *multi-armed bandit model*. Similarly to the expert setting, the costs are generated by an opponent, either non-adaptive or adaptive, and there is a maximum cost of 1 for an expert during one time turn.

The motivation for the bandit setting is that in some problems we only get to know partial information on the costs. A general class of such problems are those in which we do an experiment without being able to redo it. For example, if we have several algorithms for sorting, then in the expert

setting we get a statement "for any sequence of sequences to sort, we can have a cost comparable to the best algorithm in our disposal", *but* we need the run-times for all these algorithms, which defeats our goal. In the bandit setting, on the other hand, we need only the run-time of the algorithm we chose. As an example for how well algorithms for bandit setting perform, during a total of T turns with n experts the EXP3 algorithm [10] achieves the following performance guarantee against a non-adaptive adversary

$$\mathbf{cost}(\text{EXP3}) \leq \mathbf{cost}(\text{the best expert}) + 2.63\sqrt{Tn \ln n}.$$

Other examples of problems fitting the bandit setting are the *online routing* where we repeatedly must choose a route through a graph [47], *online advertising* where we display advertisements and hope that the user clicks the advertisement [50], and the dining problem, where we each night choose one restaurant to dine in a competitive world of food service. Note, yet again, that the assumptions on our cost generation mechanism may or may not hold in the sense that having a worst-case regret bound on all possible futures can be too restrictive (consider the dining problem above).

The existing algorithms [10, 55, 12] for the bandit setting differ from the algorithms for the expert setting in two ways.

- As the algorithms have no access to the true cost vector $\vec{c}$, they make an estimate $\hat{\vec{c}}$ of it, and then use black-box suggestions from a full-information algorithm with $\hat{\vec{c}}$ as input (perhaps with a different amount of randomization to counter the uncertainty). The estimate $\hat{\vec{c}}$ is usually the unbiased random estimate, which assigns a cost of

  $$(\text{observed cost of } \bar{\imath})/(\text{probability of choosing } \bar{\imath})$$

  to the chosen expert $\bar{\imath}$ and zero for other experts.

- In addition to the black-box advice above, the algorithms make sure that all experts are tried often enough so that the uncertainty in the estimates of the costs does not grow too significant.

  Note that in the literature this condition is achieved by sampling uniformly among the experts with a small probability $\gamma \approx \Theta(1/\sqrt{T})$, but

we have observed that the regret bounds are satisfied if any expert is selected with at least a probability γ/n. This observation potentially saves us a cost of up to $\mathcal{O}\left(\sqrt{T}\right)$ that would otherwise be incurred by sampling poorly performing experts.

It is a natural question to ask whether we can apply the idea of FPL in this bandit setting. In the following sections we answer this question; first for the case when a non-adaptive opponent generates the costs and then similarly when an adaptive opponent generates the costs.

3.2 Non-Adaptive Adversary

Recall that a non-adaptive adversary assigns costs to the experts independent from our own actions. We can generalize the FPL to work in the bandit setting, as described in Publication [P1], and this generalization is sketched in the following. With a small probability $\gamma = \Theta(1/\sqrt{T})$ we choose an expert uniformly at random. Otherwise we choose the best expert for the past *estimated* costs after these have been randomly perturbed. Hence, we choose the expert

$$\bar{i} = \arg\min_{i} \left[\!\left[\hat{\vec{c}}_{1:t-1} + \vec{\mu}_t\right]\!\right]_i ,$$

where $\hat{\vec{c}}_{1:t-1}$ is the estimated cumulative cost vector and $\vec{\mu}_t$ is the random perturbation vector. When we receive the cost $[\![\vec{c}_t]\!]_{\bar{i}}$ associated with the expert we chose, we make an estimate $\hat{\vec{c}}_t$ as described in the previous section.

We must set two unknowns: the sampling probability γ and the distribution of the perturbation. We use the following condition on the perturbation distribution and the sampling probability γ, in addition to the ones in Table 2.1. Essentially this guarantees that the probability of selecting an expert does not change relatively too much:

- If $[\![\vec{p}_t]\!]_i$ is the probability of selecting the expert i on turn t, then we must have a uniform upper and lower bound for $[\![\vec{p}_t]\!]_i \,/\, [\![\vec{p}_{t+1}]\!]_i$, not depending on T.

The regret for the bandit FPL-algorithm — we call it B-FPL— is with a properly chosen values of ϵ and γ then

$$\mathbf{cost}(\text{B-FPL}) \leq \mathbf{cost}(\text{the best expert}) + \mathcal{O}\left(\sqrt{Tn \ln n}\right),$$

where the constant inside $\mathcal{O}$-notation depends on the properties of the perturbation distribution. Note that we can set the parameters so that the decisions are identical to EXP3; this was observed by Kalai according to [30].

One of the appealing features of FPL is that it fits easily to the online geometric setting, but here we concentrated only on the simpler expert setting. There are, in fact, several known results about the geometric bandit setting, and these results naturally apply to the bandit setting with the experts. To simplify the following bounds we omit the dependence on other parameters than T. Awerbuch and Kleinberg [12] gave an algorithm that against an oblivious adversary achieves a regret of $\mathcal{O}(T^{2/3})$. McMahan and Blum [55] showed a bound for regret of $\mathcal{O}(T^{3/4} \ln T)$ against an adaptive adversary with a similar algorithm. We obtained a result that the algorithm of McMahan and Blum is $\mathcal{O}(T^{2/3})$ against a non-adaptive opponent, but later Dani and Hayes [30] tightened the bound to $\mathcal{O}(T^{2/3})$ versus the adaptive adversary, which is a strictly better result.

Although we could not give a $\mathcal{O}\left(\sqrt{T}\right)$ regret in the geometric setting, one smaller similar advantage remains for B-FPL. We do not need to assume uniform upper bounds for the costs on the experts, and we can make the regret bound smaller by taking an advantage of this. The reason is that the performance bound of B-FPL actually depends on the norm $\|\vec{c}\|_1$:

$$\mathbf{cost}(\text{B-FPL}) \leq \mathbf{cost}(\text{the best expert}) + \mathcal{O}\left(\sqrt{TM_C \ln n}\right),$$

where M_C is the upper bound to the $\|\vec{c}\|_1$. The above bound is an easy corollary to our results in Publication [P1] after we observe that uniform sampling incurs an expected cost of at most $\gamma M_C/n$ during one time turn.

One unfortunate drawback remains. If we want a regret of the order $\mathcal{O}\left(\sqrt{T}\right)$, then we need the probabilities of selecting the experts, and these

are not necessarily easy to obtain. For example, for the two-sided exponential distribution we need to calculate several integrals to do this.

3.3 Adaptive Adversary

This section considers a bandit setting in which our own actions can affect the future costs, i.e., we play against an adaptive opponent. The bandit FPL algorithm does not guarantee good regret against an adaptive opponent, as we can see from the following sketch of an example that was given in [30].

Example: Bandit algorithms against an adaptive opponent. *For two experts, A and B, we make a cost generation mechanism where the variance of the cost estimate of the expert A is high, so that the real cost may hide under the variance. Let p_A be the probability of selecting the expert A. Then choose $c_A = 0$, if $p_A < \Theta(1/\sqrt{T})$, and otherwise $c_A = 1$. Set $c_B = 1 - c_A$, i.e., the opposite of the cost of the expert A. The probability p_A is usually of the order $\Theta(1/\sqrt{T})$ and Dani and Hayes note that while the expected regret versus both experts is $\mathcal{O}\left(\sqrt{T}\right)$, the variance of the regret of the expert A is $\Omega(T^{2/3})$. Hence the expected maximum of the regrets is $\Omega(T^{2/3})$.*

As the problem in this example is that variance makes our cost estimate too high, we check whether we can alleviate this by making sure that we do not overestimate the costs. In fact something similar was done by Auer [9], though to obtain regret bound with high probability, not specifically proving a bound against an adaptive opponent. Later, Cesa-Bianchi and Lugosi [24] observed that the algorithm works against an adaptive opponent. The trick was to add an additional additive factor proportional to $1/p_i$ to the cost of the ith expert, which guarantees that the cost of the expert is almost never over-estimated.

The factor $1/p_i$ is a linearization of what is subtracted in an algorithm in Publication [P2]: a term upper bounding the standard deviation of the estimate of the cost. Formally the subtracted term $[\![\hat{\sigma}_t]\!]_i$ from the unbiased

estimate $\left[\!\left[\hat{\vec{c}}_{1:t}\right]\!\right]_i$ is

$$[\![\hat{\sigma}_t]\!]_i = \underbrace{(1+\frac{1}{n})\sqrt{\lg t}}_{\text{a small value}} \qquad \underbrace{\sqrt{\sum_{\tau=1}^{t}\frac{1}{p_{i,\tau}}}}_{\substack{\text{a upper bound to} \\ \text{standard deviation of } \left[\!\left[\hat{\vec{c}}_{1:t}\right]\!\right]_i}} \qquad .$$

We use a seldom-used concentration bound to show formally that using $\hat{\sigma}_t$ results in an estimate which is almost never more than the real cost, i.e., $\hat{\vec{c}}_{1:t} - \hat{\sigma}_t \leq \vec{c}_{1:t}$ with high probability. Of course, we must make sure that the cost estimate is not too small, because then the bias $\left\|\vec{c}_{1:t} - (\hat{\vec{c}}_{1:t} - \hat{\sigma}_t)\right\|$ would be too large, and the increased regret would make this approach useless. Fortunately we were able to show that the bias does not grow too large for this to happen.

The resulting regret bound for a two-sided exponential distribution is

$$\mathcal{O}\left(\sqrt{n\,T\ln T}\right),$$

which matches the previous bound.

Chapter 4

On the Competitive Analysis of BSTs

This chapter proceeds to an application of the competitive analysis: the performance of data structures and more precisely ones that are based on binary search trees (BSTs).

4.1 Background on the BSTs

BSTs are a fundamental class of data structures that solve the DICTIONARY problem for items with an order relation. They also support operations PREDECESSOR and SUCCESSOR and are also rather amenable to adding other operations. In the DICTIONARY problem we store a (dynamic) set of items and support operations SEARCH, INSERT, and DELETE for the items that may have associated data. More extensive details are found in the standard textbooks [48, 28].

In abstract terms a BST specifies an order in which we perform the comparisons to limit the number of potential items that we search (and we use only comparisons to do this). The search begins with a comparison at the *root* of the BST and continues recursively until the searched item is found. This process of comparisons is associated with a search tree of nodes, where each node corresponds to a comparison.

When we use BSTs and want to guarantee good performance, we may keep the BST approximately balanced, like in red-black trees (RB-trees) [28], because they guarantee a $\Theta(\lg n)$ worst-case cost per operation. However, RB-trees change only when updated and it might be that the sequence of accesses has some kind of structure which potentially could be exploited (like in the MTF-rule for lists on page 11). For example, iterating over items in a sorted order is a typical operation with a very specific structure. A more general regularity are the *working sets*, which intuitively mean that operations are connected either spatially or temporally [33]. In spatial connectivity the items that are close to each other are more likely to be accessed together (like names that share a prefix). Similarly, in temporal connectivity an item that was accessed in recent past is also more likely to be accessed in close future. In Sections (4.5 – 4.7) we study particular questions on BST algorithms for these and even more general regularities.

However, first in Section 4.3 we are interested in a simpler regularity, empirical frequency of items, which is the frequency that an item occurs in a sequence of the accesses. Of course, some items may be frequently accessed, and some rarely or never, for example in a symbol table of a compiler. An important point is that we will not assume that we know the frequencies a priori.

4.2 Counting Costs in BSTs

Before we continue, we discuss the issue of the cost of the operations in BSTs. It is complicated to model realistically, and it depends both on the platform and the size of the data. We recapitulate these issues, because they are important to keep in mind.

The natural model with BSTs is the *comparison model*, which defines the cost of an access as the number of comparisons performed before finding the correct item or its non-existence. In it we assume that the search relies only on the order relation of the items and not on their bit representation.

In the comparison model the lower bound for a search is $\lg n$ as in the worst-case each comparison may leave half of the remaining candidates to

be searched. This lower bound does not hold in a more general model of cost, because Fredman and Willard [37] showed that at least in theory it is possible to obtain $\mathcal{O}(\lg n/\lg\lg n)$ cost in the *unit cost model*, where we can rely on other techniques than comparison to speed things up. By "other techniques" we mean using the binary presentation of the items. Later, Andersson et al. [7] noted that with slightly super-linear space $O(\sqrt{\lg n})$ cost is achievable with a similar technique. These are superior in $\mathcal{O}$-notation, but unfortunately the algorithms are complex and in Fredman and Willard's approach the height of the resulting tree is $5\lg n/\lg\lg n$ so they have not resulted in practical applications (so far).

Another approach is the so-called van Emde Boas tree [68], which searches over the binary presentation of the key, and for w bits in a key the search costs $\mathcal{O}(\lg w)$. This idea has been, for example, applied to derive a specialized algorithm to serve operations during the Internet protocol routing [31]. Unfortunately, for small data sets the space usage of the van Emde Boas tree can be high.

In this Thesis we essentially use the comparison model. Hence, the operations we use during a search are simple: comparisons and following pointers. However, we actually equal the cost with the number of nodes that are accessed (including both the root and the item). We allow a constant number of comparisons in a single node, because the cost remains the same in the $\mathcal{O}$-notation and this simplifies our analysis. We set a cost of k for restructuring a sub-tree of size k, which is asymptotically a realistic assumption, because in the unit cost model the cost is $\Theta(k)$ [54]. Also, when the nodes are at random positions in the memory, these costs should approximately equal the number of cache misses or page faults, unless the history from the previous searches interferes.

4.3 Efficient Use of FPL with BSTs

After discussing the issues with costs, we can continue to how we can efficiently exploit empirical frequencies with BSTs. One method for this is the recently (re-)introduced FPL algorithm which has an application to

Input:

- A set S of items.
- A hidden sequence $\sigma = \langle \sigma_1, \dots, \sigma_T \rangle$ of accesses to S that will be served online.
- An algorithm OPT-BST that computes an optimal BST for the access counts so far.

State: items are stored in a BST. Additionally, the algorithm stores a count c_i of accesses and a random number r_i for each item i.
Initialization: initialize each r_i independently to a random uniform number between 1 and $\sqrt{T/n}$. Built a BST using OPT-BST algorithm with input from perturbed counts $c_i + r_i$.
Serving the access to σ_i: search the item σ_i as usual. Update the access count c_i for the item σ_i and *if* this count becomes 0 modulo $\sqrt{T/n}$ then globally rebuild the whole BST using OPT-BST algorithm with input from the perturbed counts $c_i + r_i$.

Table 4.1: The online BST-FPL algorithm as in [47].

BSTs [47]. The application is the BST-FPL algorithm in Table 4.1. The main point of the algorithm is to occasionally compute the optimal BST for the past frequencies after these have been randomized.

What is interesting in BST-FPL is that, for any possible access sequence, it can serve it almost as well as the best possible BST for that sequence *in hindsight*. More formally, let the cost be as in the previous section, then for any sequence σ with T accesses and n items the cost of serving σ with BST-FPL is at most

$$\textbf{cost}(\text{BST-FPL}\,(\sigma)) \leq \textbf{cost}(\text{the best static BST for } \sigma) + 2n\sqrt{nT}. \qquad (4.1)$$

Hence, no matter how we choose the frequencies of the items, the average access cost will be close to the optimal, even if we would have known these

Input: items $\{1, \ldots, n\}$ and probabilities p_i for each item i.
Output: A BST that minimizes the expected path length

$$\sum_{i=1}^{n} p_i \, d_i,$$

where d_i is the number of nodes on the path from the root to the item.

Table 4.2: Requirements for OPT-BST algorithm.

frequencies and chosen a BST for them. Kalai and Vempala call this *strong static optimality*, whereas previously, for example, the splay tree of Sleator and Tarjan [65] achieved *static optimality* that means a convergence with a multiplicative constant factor, i.e.,

$$\mathbf{cost}(\cdot) \leq \mathcal{O}(\mathbf{cost}(\text{the best static BST for } \sigma)),$$

which in data structures can be a significant difference. The difference is significant because BSTs are a frequently needed solution, so their run-time is important; also the constant factor determines the costs to a large extent for many values of n, because the costs usually scale in $\mathcal{O}(\lg n)$.

The upper bound (4.1) shows that in theory it is possible to achieve excellent performance in this case. Note that the average cost between BST-FPL and the best BST tends to zero as T increases, hence the performance of BST-FPL is in some sense optimal. An unfortunate aspect in BST-FPL is that we must do global rebuild approximately every $\sqrt{T}$th access (amortized) with the OPT-BST algorithm that satisfies the conditions in Table 4.2. The cost of doing these is not counted in Equation 4.1.

Unfortunately the current best algorithm for the global rebuild with n items takes both $\Theta(n^2)$ time and space [48]. Hence, because there are T accesses and approximately $\sqrt{T}$ rebuilds, the average cost incurred by rebuilds is $\Theta(n^2\sqrt{T}/T)$ in *time* (in instructions). This implies that in order to reduce the amortized rebuilding costs to $\mathcal{O}(1)$ the T must be $\Theta(n^4)$, and even then we must occasionally use $\Theta(n^2)$ space. Thus we definitely would

like to avoid the additional cost associated with the global rebuilds and, indeed, there are $\mathcal{O}(n)$ space and either $\mathcal{O}(n)$ or $\mathcal{O}(n \lg n)$ time approximate solutions to the above construction problem [48, 45, 36, 56]. In fact one such algorithm is proposed in this Thesis, namely the AWOBST algorithm, the details of which are given in Chapter 5 and which has the advantage of having a low I/O-cost.

However, when we use an approximately optimal BST then the analysis behind the upper bound (4.1) fails. Intuitively the difference between using an optimal or an approximation algorithm does not appear to be significant. If we use an approximate BST, we would expect to compare our own performance to the performance of the approximate BST that has been formed in hindsight. That is, we would like to have a bound that resembles something like this:

$$\mathbf{cost}(\text{BST-FPL}\,(\sigma)) \leq \mathbf{cost}(\text{approximately optimal BST for } \sigma) + 2n\sqrt{nT},$$

where we have replaced the optimal BST in the upper bound (4.1) with an approximate BST. In the following section we note that most approximation algorithms achieve a bound that is close to the above bound.

4.4 Approximate BSTs Work with FPL

This section is more technical than the rest of the introduction to this Thesis. We give a theoretical justification on using an approximate BST with FPL. This is previously unpublished work and not completely trivial. First, for completeness, we give the following example, by Kakade et al. [46].

Example: Where FPL fails with an approximate oracle. *We have n experts and an approximate oracle* APPROX*, which selects an expert such that*

$$\mathbf{cost}(\textit{expert chosen by } \text{APPROX}) \leq \mathbf{cost}(\textit{best expert}) + f(n),$$

where $f(n)$ is the additive error of APPROX*. We now run* FPL *for T turns and we replace the optimal oracle* OPT *in* FPL *with* APPROX*, i.e., we may*

now choose approximately optimal decisions for past perturbed costs. To obtain a large regret for the first $f(n)$ turns let the first expert have a cost of one and rest zero. Then for $f(n)$ turns let the second expert have the cost of one and rest zero. This is repeated until the nth expert has finished, at which point we may iterate and select the first expert again. Note that during each turn APPROX *may select any expert, because the difference in the costs of any two experts is less than $f(n)$ and so the definition of* APPROX *does not limit its output. Hence we may suffer a cost of one every turn if we use* APPROX*, but all experts have a cost that on average is less than $1/n$ per time turn. Thus, the additive error $f(n)$ of* APPROX *does not limit the regret of* FPL *using* APPROX*.*

In general we do not know how to remove this hindrance in FPL (such a method exists for the so called Zinkevich's algorithm [76], see [46]). However, we can obtain a good bound, if we assume additional properties on the part of the APPROX, as noted in [46]:

Theorem 1. *If for an approximate oracle* APPROX: $S_C \to S_D$ *and for any cost vector $\vec{c} \in S_C$ we have that that each coordinate $[\![\text{APPROX}(\vec{c})]\!]_i$ in the vector* APPROX$(\vec{c})$ *is bounded separately, i.e.,*

$$[\![\text{APPROX}(\vec{c})]\!]_i \leq [\![\text{OPT}(\vec{c})]\!]_i + f(n),$$

then the additive approximation error applies as such to the online regret of the FPL *using* APPROX*, i.e.,*

$$\mathbf{cost}(\text{FPL } \textit{with } \text{APPROX}) \leq \mathbf{cost}(\text{FPL}) + T\,f(n).$$

For BSTs the output of APPROX is a vector that lists the depths of the items. Unfortunately, the condition of the above theorem does not hold for these BST vectors in the sense that $f(n) \approx \lg n$ is the best possible bound. To see why, generate the accesses from a uniform distribution, then some item is the root in the optimal BST, but could be down at depth $\lg n$ in the approximate BST.

Fortunately, in the particular case of BSTs we can show that nearly all approximation algorithms for optimal BSTs also give a good bound in the

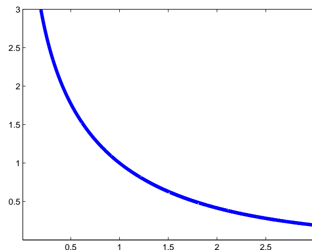

Figure 4.1: An illustration of a part of the decision set $\mathbb{S}^2$ which consists of the vectors on the drawn line.

online setting. Intuitively, what we do is observe that the vector space of BSTs has a certain shape $\mathbb{S}$, and it is possible to show that most approximation algorithms always choose BSTs close to the optima in $\mathbb{S}$. Additionally, the performance of the optimal BST and the optimal vector in $\mathbb{S}$ have similar performance, though they may be far apart in the norm. These two reasons, together with the fact that the width of the part of $\mathbb{S}$ we use is bounded, are enough to obtain a good bound in the online setting. We give a formal presentation in Theorem 2 below, but first we actually define $\mathbb{S}$. The subset $\mathbb{S}^n$ of $\mathbb{R}^n$ is

$$\left\{ \langle -\lg p_1, \ldots, -\lg p_n \rangle \in \mathbb{R}^n \,\middle|\, \sum_{i=1}^{n} p_i = 1 \text{ and } 0 \leq p_i \leq 1 \text{ for each } p_i \right\}.$$

We use $\mathbb{S}$ to refer to $\mathbb{S}^n$ when the dimension n is clear from the context, also see the illustration of the set $\mathbb{S}^2$ in Figure 4.1. The set $\mathbb{S}$ relates to the information-theoretic properties of lengths of code words, which in turn relate to path lengths in BSTs. To relate $\mathbb{S}$ and BSTs we need the following definition of entropy which, intuitively, is a measure of randomness of a distribution, because it approximates the amount of bits needed to encode samples from the distribution [29].

Definition 1. *For a probability distribution $\vec{p} = \langle p_1, \ldots, p_n \rangle$ on items*

$\{1, \ldots, n\}$ *the entropy* $H(\vec{p})$ *of the distribution* $\vec{p}$ *is*

$$\sum_{i=1}^{n} -p_i \lg p_i.$$

We use the shorthand H for $H(\vec{p})$ when $\vec{p}$ is clear from the context. The entropy is related to BSTs through the following bounds [6, 29].

$$H(\vec{p}) - \lg H(\vec{p}) - \lg e - 1 \leq \mathbf{E}_{\vec{p}}(\mathbf{cost}(\text{optimal BST for } \vec{p})) \leq H(\vec{p}) + 1. \quad (4.2)$$

We now proceed to the theorem that shows the competitiveness of approximate BSTs with BST-FPL.

Theorem 2. *Let* APPROX *compute, for input probabilities* $p_1, \ldots, p_n$ *on the items, an approximately optimal* BST *in which any item* i *is at depth* $-\lg p_i + c$ *or less. Then if we use* APPROX *instead of the optimal algorithm* OPT *in* BST-FPL*, the bound for any sequence* σ *of* T *accesses is*

$$\begin{aligned} & \mathbf{E}(\mathbf{cost}(\text{FPL } \textit{using} \text{ APPROX } \textit{on } \sigma)) \\ = \; & (\textit{upper bound of an approximate} \text{ BST } \textit{for } \sigma) + 2\sqrt{nT} \lg \left(T + n\sqrt{T}\right). \end{aligned}$$

Proof. Fix any access sequence σ. From now on the input to APPROX consists of access counts, not access probabilities; we can change between these two because access counts give empirical frequencies. The output is the BST in a vector format, i.e., a vector that lists the depths of the items. To serve σ, we use the BST-FPL with the APPROX and we want to bound the expected cost

$$\mathbf{E}\left(\sum_{t=1}^{T} \text{APPROX}(\vec{c}_{1:t-1} + \vec{\mu}_t) \cdot \vec{c}_t\right),$$

where, as in Section 4.3, the cost vector $\vec{c}_t$ is an indicator vector; in other words, the coordinate corresponding to the accessed item is one and the other elements are zero. We also define $\vec{c}_{1:t-1}$ as $\sum_{\tau=1}^{t-1} \vec{c}_\tau$.

The upper bound on the performance of APPROX is

$$\text{APPROX}(\vec{c}_{1:t-1} + \vec{\mu}_t) \cdot \vec{c}_t \leq -\lg [\![\vec{p}_t]\!]_{\bar{i}} + c, \tag{4.3}$$

where $\bar{i}$ is the access that is indicated by $\vec{c}_t$ and $[\![\vec{p}_t]\!]_i$ is the empirical frequency of the ith item as implied by counts in the perturbed cost vector $\vec{c}_{1:t-1} + \vec{\mu}_t$. Let us now compare this to the cost of a FPL algorithm that uses vectors in $\mathbb{S}$ as decision vectors and has the same input:

$$\text{OPT}(\mathbb{S}, \vec{c}_{1:t-1} + \vec{\mu}_t) \cdot \vec{c}_t,$$

where by $\text{OPT}(\mathbb{S}, \vec{c}_{1:t-1} + \vec{\mu}_t)$ we denote that the decision vectors come from $\mathbb{S}$, and so $\text{OPT}(\mathbb{S}, \vec{c})$ outputs a vector $\vec{x}$ in $\mathbb{S}$ that minimizes $\vec{x} \cdot \vec{c}$. We can define $\text{OPT}(\mathbb{S}, \vec{c})$ as a function of the frequencies $[\![\vec{p}_t]\!]_i$; the $\text{OPT}(\mathbb{S}, \vec{c})$ is $-\lg \vec{x}$, in which the vector $\vec{x}$ is the minimum of the function

$$f_{\vec{p}}(\vec{x}) = \sum_{i=1}^{n} - [\![\vec{p}]\!]_i \lg [\![\vec{x}]\!]_i \text{ given that } [\![\vec{x}]\!]_i \geq 0 \text{ and } \sum_{i=1}^{n} [\![\vec{x}]\!]_i = 1.$$

Now recall a classic result in the information theory that the minimal vector $\vec{x}$ equals the empirical frequency vector $\vec{p}_t$. Hence, we get

$$\text{OPT}(\mathbb{S}, \vec{c}_{1:t-1} + \vec{\mu}_t) \cdot \vec{c}_t = -\lg [\![\vec{p}_t]\!]_{\bar{i}}\,. \tag{4.4}$$

Comparing Equations (4.3) and (4.4) we obtain an upper bound for cost of BST-FPL that uses the approximation algorithm APPROX instead of the optimal oracle:

$$\mathbf{E}\left(\sum_{t=1}^{T} \text{APP}(\vec{c}_{1:t-1} + \vec{\mu}_t) \cdot \vec{c}_t\right) \leq \mathbf{E}\left(\sum_{t=1}^{T} \text{OPT}(\mathbb{S}, \vec{c}_{1:t-1} + \vec{\mu}_t) \cdot \vec{c}_t\right) + cT.$$

To upper bound the right-hand side of the above inequality we can use the performance bound of FPL (we do not fix M, M_C, and M_D yet):

$$\mathbf{E}\left(\sum_{t=1}^{T} \text{OPT}(\mathbb{S}, \vec{c}_{1:t-1} + \vec{\mu}_t) \cdot \vec{c}_t\right) \leq \text{OPT}(\mathbb{S}, \vec{c}_{1:T}) \cdot \vec{c}_{1:T} + 2\sqrt{M\, M_C\, M_D\, T}.$$

Recall the definitions of parameters M, M_C, and M_D:

- M is the maximum cost on a single turn.
- $M_C = \max_{\vec{c} \in S_C} \|\vec{c}\|_1$, where S_C is the set of cost vectors.
- $M_D = \max_{\vec{d_1}, \vec{d_2} \in S_D} \|\vec{d_1} - \vec{d_2}\|_1$, where S_D is the set of decision vectors.

At first it might appear that we can not bound these values. For example, the distance between two decisions M_D, can be arbitrarily large as the width of $\mathbb{S}$ is infinite, because $\lim_{x \to 0} - \lg x = \infty$. However, when we fix T, then OPT chooses only parts of $\mathbb{S}$. This is because we can think that the perturbation places at least one access to each item and then we can bound the empirical probability $[\![\vec{p}]\!]_i$ of accessing an item i from below. For T real accesses the perturbation places at most $n\sqrt{T}$ accesses, and hence for each item's empirical frequency it holds that

$$[\![\vec{p}]\!]_i \geq \frac{1}{T + n\sqrt{T}} = \frac{1}{T'}.$$

Here we introduced notation $T' = T + n\sqrt{T}$, which is very close to T. Now for any vector $\vec{d} \in S$ which OPT may choose, we have that $\vec{d_i} = -\lg [\![\vec{p}]\!]_i \leq \lg T'$.

This implies that we can use the following values for the parameters:

- $M = \lg T'$.
- $M_C = 1$.
- $M_D = n \lg T'$.

Finally, we can bound $\text{OPT}(\mathbb{S}, \vec{c}_{1:T}) \cdot \vec{c}_{1:T}$ using Inequality 4.2:

$$\text{OPT}(\mathbb{S}, \vec{c}_{1:T}) \cdot \vec{c}_{1:T} \leq \textbf{cost}(\text{the best BST for } \vec{c}_{1:T}) + T(\lg \lg n + \lg e + 1 + c),$$

which proves the claim. □

The above theorem is slightly better than what we aimed to: a bound that is worse than the upper bound (4.1) only by the approximation factor. This is possible because with the optimal oracle the worst-case cost was set to n, which happens if the BST is a list. With the approximation algorithm the bound on the approximation makes this unlikely for almost all values of T. Unfortunately this result relies on specific properties of BSTs and as such does not give a general result on how to use approximate oracles with FPL.

4.5 Offline BST Algorithms

Recall that in the competitive analysis we compare the cost of an online algorithm to an algorithm with knowledge of the future accesses in the sequence σ:

$$\mathbf{cost}(\text{online BST algorithm on } \sigma) \leq \mathcal{O}(\mathbf{cost}(\text{best offline algorithm on } \sigma)) .$$

In previous sections the offline algorithm was limited to a static BST, but what if we want more than performance comparable only to the best BST in hindsight? The best static BST in hindsight depends only on the empirical frequencies of items in σ, and hence it can miss more "dynamic" regularities that depend on the ordering of the operations in σ.

One way to capture the more general regularities is to compare our online BST algorithm against a more powerful offline BST algorithm than the static BST. One choice is an offline BST algorithm that is not limited in its capabilities. Thus, offline BST algorithms may also, at a cost, change the structure of their internal BST between the accesses. *Dynamic optimality* is the term for achieving a performance comparative to the offline BST algorithms, save for the multiplicative constant factor [65]. Hence, the difference to the static optimality is that the comparator can change during the access sequence, at a cost. Table 4.3 lists the definitions of different types of optimalities for BST algorithms.

It is interesting to know what are the regularities that we can capture with offline BST algorithms. Remarkably many of the upper bounds are

optimality	cost must be less than
static	$\mathcal{O}(\mathbf{cost}(\textsc{bst}_{\text{static}}))$
strong static	$\mathbf{cost}(\textsc{bst}_{\text{static}}) + o(T)$
dynamic	$\mathcal{O}(\mathbf{cost}(\textsc{bst}_{\text{dynamic}}))$
almost dynamic	$\mathcal{O}(\lg \lg n \, \mathbf{cost}(\textsc{bst}_{\text{dynamic}}))$

Table 4.3: Definitions of different types of optimalities for BST algorithms. Here $\textsc{bst}_{\text{static}}$ is any static BST storing n items, and $\textsc{bst}_{\text{dynamic}}$ is any BST algorithm that can change the BST during the access sequence. The cost bound must hold for any access sequence $\sigma = \langle \sigma_1, \ldots, \sigma_T \rangle$ with more than n accesses.

constructive and proved for the splay tree of Sleator and Tarjan [65], which is an online self-adjusting data structure. It is a long-standing open problem to show whether the splay tree achieves dynamic optimality. On the other hand, there are several known results, and we now list some of them. We assume that the number of accesses is more than n to simplify the results.

- The splay tree is statically optimal, i.e., its cost is always

  $$\mathcal{O}(\mathbf{cost}(\text{best static } \textsc{bst})) \, .$$

  Compare the static optimality to the strong static optimality in Section 4.3, which replaces the multiplicative constant factor with a small additive term.

- The splay tree captures time dependent regularities, working sets, because the cost of the splay tree is

  $$\mathcal{O}\left(\sum_{\sigma_t \in \sigma} \lg \left(\text{number of accessed distinct items since the last access to } \sigma_t\right) \right) ,$$

  and hence the cost depends on the temporal reference of locality in the sequence σ.

- Spatial locality of reference, on the other hand, is captured by the static finger theorem [65], which says that the amortized cost of accessing σ_t is $\mathcal{O}(\lg \mid \sigma_t - i \mid)$ for any fixed i, and the dynamic finger theorem which gives an amortized cost of $\mathcal{O}(\lg \mid \sigma_t - \sigma_{t-1} \mid)$ [27, 26]. Another such result is the scanning theorem that shows that iterating the items is a $\mathcal{O}(n)$ time operation [66].

Despite these results we note that the splay tree is not necessarily the best solution, because its performance can be slightly worse than that of other solutions [74] and in experiments the splay tree performs better only when there is a structure in the accesses [60]. Thus, static optimality does not imply good performance for random or non-dynamic data, because the $\mathcal{O}$-notation has a constant factor in it and thus balanced BSTs are faster in practice for such data [60]. Note that Albers and Karpinski [4] and Fürer [40] have proposed that the lack of performance is caused by unnecessary operations that change the structure of the BST during a search. Another explaining factor, as pointed out by Aho et al. [1], is that the BST in the splay tree is in practice unbalanced when compared for example to RB -trees (which in experiments appear to be on average almost perfectly balanced for uniform accesses).

In the next section we consider limits on how well the offline BST algorithms can perform and in Section 4.7 we consider how to implement an online BST algorithm with a formal guarantee against offline BST algorithms.

4.6 Limits of BST algorithms

Even though it is unrealistic to foretell what are the future accesses to a BST, the limits on the performance of offline BST algorithms are interesting, because they also give a theoretical bound on how well we can perform online. Also, they can give insights on how to implement BST algorithms, such as in [32].

Wilber [73] was the first one to give lower bounds for offline BST algorithms. He gave in fact two different lower bounds, which both depend on the access sequence and neither of which is simple. Hence, we give only

an application of the first of the bounds to a sequence generated from a random source, because this relates to results in Publication [P3].

We are given a sequence $\sigma = \langle \sigma_1, \ldots, \sigma_t \ldots, \sigma_T \rangle$. We assume that we generate each item σ_t in σ independently and identically (i.i.d.), which means that σ_t is the item i with probability p_i, and these probabilities are the same for all t. *If* we serve this kind of sequence with a static BST *and* place items only to leaves, not internal nodes, then the expected lower bound is the entropy of the generating distribution, because of a classic result which lower bounds the expected length of a code for a distribution [29]. However, if a BST algorithm is used then it is not clear what the lower bound is, because "the code" could change on the fly. Wilber gave the following theorem that relates the cost of BST algorithm and the entropy of the random source producing the sequence.

Theorem 3 (Wilber [73]). *Let us generate a sequence $\sigma = \langle \sigma_1, \ldots, \sigma_t, \ldots, \sigma_T \rangle$ from a random source where each access σ_t is generated according to probabilities $\mathbf{P}(\sigma_t = i) = p_i$. The entropy of the source $H(\vec{p})$ is $\sum_{i=1}^{n} -p_i \lg p_i$ and*

$$\mathbf{E}_\sigma \left(\mathbf{cost}(\textit{any BST algorithm serving } \sigma) \right) \geq \frac{T\,H\left(\vec{p}\right)}{3} = \Omega\left(T\,H\left(\vec{p}\right)\right).$$

It is not clear how tight this lower bound is, apart from the fact that the best static BST serving σ has a upper bound of $H(\vec{p}) + 1$ for the expected cost of an access. Hence, Theorem 3 implies that any offline BST algorithm can only be a constant factor faster than a balanced BST when the accesses are random.

Why is this result interesting? It asserts that, if we assume that our accesses behave i.i.d., then essentially there is no reason to use anything more complicated than static BSTs, because the overhead in more complicated schemes is likely to be too large. In this case we limit our input (instead of limiting it to a subset we define the process that generates the input). Usually the bounds in the competitive analysis hold over all possible inputs, but for example Angelopoulos et al. [8] argue that in some applications we must assume additional properties on the input to derive

relevant results. Intuitively, this is so because those input sequences, which force the competitive ratio of an online algorithm high, may be rare.

However, neither of the assumptions that Wilber makes, the independence and identical distribution, may hold. This is because the sequences may have many kinds of regularities — like in working sets — and these definitely can cause the accesses to depend on each other. We can, of course, broaden the assumptions on the source that produces σ. The most general stochastic source gives out sequences σ; i.e., for each σ there is an associated probability $\mathbf{P}(\sigma)$. Then the entropy is

$$H = \sum_{\sigma} -\mathbf{P}(\sigma) \lg \mathbf{P}(\sigma) \,,$$

which is counted over *sequences*, not over individual items. In Publication [P3] we give the following result [1].

Theorem 4. *Let us generate a sequence $\sigma = \langle \sigma_1, \ldots, \sigma_t, \ldots, \sigma_T \rangle$ from a random source, where each sequence σ is assigned a probability $\mathbf{P}(\sigma)$. The entropy H is $\sum_{\sigma} -\mathbf{P}(\sigma) \lg \mathbf{P}(\sigma)$ and*

$$\mathbf{E}_{\sigma} \left(\mathbf{cost}(\textit{any}\ \textrm{BST}\ \textit{algorithm on}\ \sigma) \right) \geq \frac{H}{6} = \Omega(H).$$

This generalizes Theorem 3. Note that if the data is i.i.d., then H in the above theorem equals $T\,H(\vec{p})$, where $H(\vec{p})$ is entropy of a source generating a single access.

Is there a matching upper bound as there was in the i.i.d. case? The answer is negative, because if the source produces a fixed sequence then its entropy H is zero, but the cost of a fixed sequence can be $\Omega(T \lg n)$ [73]. Nevertheless, using Theorem 4 we can show that the splay tree is optimal when the sequence σ is generated from a Markov chain source with a spatial locality of reference. By this we mean that

$$\mathbf{P}(\sigma_t = i) \leq 1/\sqrt{1 + \mid i - \sigma_{t-1} \mid}.$$

[1] In Publication [P3] we assume that the accessed item is assigned to the root. However, the proof generalizes with minor modifications to cases where this is not true.

Additionally, in Publication [P3], we gave a result that in a certain sense links the complexity of the sequence σ to the cost of the BST algorithm. Intuitively, if the access sequence σ is complex, then the cost of BST algorithm serving it should be high. We, of course, have the problem of defining what "complex" means. We use Kolmogorov complexity [51], which intuitively defines the complexity of a sequence σ as the length of the shortest computer program printing it (formally the computer program is given for a universal Turing machine). We give a bound

$$\mathbf{cost}(\text{any BST algorithm serving } \sigma) \geq \frac{(\text{Kolmogorov complexity of } \sigma)}{6}.$$

This bound is far from optimal, as Kolmogorov complexity is low on many sequences, such as pseudo-random sequences, which are nevertheless expensive to serve.

In addition to these and Wilber's bounds, the only other work on lower bounds that we are aware of is in Harmon's Ph.D. Thesis [43], where he gives an injective mapping from BST algorithms serving a specific sequence to points in a $n \times T$ grid, where the cost of the BST algorithm is the number of the points. Using this mapping he gives a family of bounds, which includes those of Wilber.

4.7 Augmenting BSTs to Become Dynamic

In the previous section we were interested in lower bounds on the cost of offline BST algorithms. These algorithms are dynamic in the sense that during the access sequence the BST can change. Now we turn to a related topic of providing upper bounds, and preferably giving explicit constructions.

The history of such bounds is quite short, because the first (and basically only) algorithm achieving a non-trivial bound is the algorithm Tango by Demaine et al. [32]. The subsequent work, like [72] and ours, is primarily based on the idea behind Tango. The idea is to use a lower bound on the offline BST algorithms and show that for each unit of cost suffered by an offline BST algorithm, Tango suffers a cost of the order $\mathcal{O}(\lg \lg n)$.

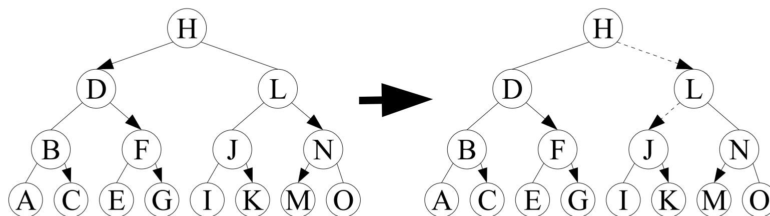

Figure 4.2: The P-tree on the left illustrates the final state of arrows in a P-tree that has served a sequence that consists of accesses in order H, L, J, E, O, I, A, C, K, N, M, B, F, D, and G. The P-tree on the right illustrates how the arrows in the P-tree on the left change when the item K is accessed, the dashed line in the arrow indicates a switched arrow.

Hence the cost of Tango is at most $\mathcal{O}(\lg \lg n)$ times the cost of any offline algorithm. Recall that dynamical optimality of an online BST algorithm means that the algorithm costs at most $\mathcal{O}(1)$ times the cost of any offline algorithm.

4.7.1 Wilber's lower bound for BST algorithms

We now go through the lower bound, as we need it to understand both Tango and our work. Assume that the items form a perfectly balanced binary search tree, which we call a P-tree (note that this requires exactly $2^x - 1$ items for some $x \geq 0$). Serve an access sequence σ with the P-tree. Associate to each node in the P-tree a two-state arrow that points to the direction of the child whose sub-tree was most recently accessed. When serving σ count the number of times these arrows switch. This count plus $n/2$ is a lower bound to the cost of any BST algorithm.

See an illustration of an access sequence and the final state of the P-tree in Figure 4.2. Originally this lower bound was given by Wilber [73], but Demaine et al. [32] gave this revised description.

4.7.2 Tango: the almost competitive BST algorithm

Let us also sketch the Tango algorithm so that we can discuss its shortcomings. Tango is conceptually a tree of balanced BSTs. After serving an access σ_{t-1}, each of these sub-BSTs include items on a continuous path of arrows in the P-tree. That is, consider nodes from a leaf towards the root, and add the parent of the node to a BST until the arrow of the parent does not point to the node under consideration. The maximum number of items in a sub-BST is $\lg n$ since it contains the items on a path and the root-to-leaf path in the P-tree is of length $\lg n$. When serving an access σ_t, each switch of an arrow in the P-tree makes Tango perform computation with a cost of $\mathcal{O}(\lg \lg n)$. The intuitive reason is that for each switch of an arrow in the P-tree Tango operates on a sub-BST with at most $\lg n$ items. Hence, Tango is $\mathcal{O}(\lg \lg n)$-competitive, as the number of the switches of the arrows is a lower bound to the cost of any offline BST algorithms.

Publication [P4] addresses the following shortcomings in this construction.

- The worst-case cost in Tango is $\mathcal{O}(\lg n \lg \lg n)$ which is higher than the usual $\mathcal{O}(\lg n)$.
- Tango does not support updates to the tree.

Also we wanted to study, in theory, the feasibility of this approach. Note that a more recent algorithm — the multisplay tree [72] which is also based on idea behind Tango — supports amortized $\mathcal{O}(\lg n)$ search, and updates in $\mathcal{O}(\lg^2 n)$ time. Unfortunately, although in this approach the P-tree is explicitly stored to the nodes of the tree, it is not used during the searches, which appears wasteful.

4.7.3 Our approach for a competitive BST algorithm

We now give the basic idea of the approach, called Poketree, which is given in Publication [P4]. We do the searches directly on a balanced search structure that mimics the P-tree. This balanced search structure could be a RB-tree or a deterministic skip list [62, 58]. We can see these structures

as perfectly balanced search trees because they are implementations of a B-tree for the value $B = 3$, i.e. they are 2-3-4-trees, and B-trees are always perfectly balanced. The normal static search on the balanced structure alone does not guarantee $\mathcal{O}(\lg \lg n)$ -competitiveness, so we need a way to search quickly if the accessed item is on a path formed by many consecutive arrows in the P-tree. This search is essentially a binary search on its own. We can implement this binary search in several ways, but note that we can implement the search with only one additional pointer at each node. Thus, our cost model does not break down for doing too much computation in one node. We call this pointer a *dynamic pointer* and it essentially points to the sub-tree that was most recently accessed. We illustrate this search structure in Figure 4.3.

The search in Poketree proceeds as follows. First try the additional dynamic pointer. Note that we do not have to follow it, if we cache the minimal and maximal item in the sub-tree that the pointer points to. If the dynamic pointer fails, then we follow a usual child pointer supplied by the underlying balanced search structure. We can update the dynamic pointers after we find the item we searched for by backtracking along the same nodes (which is a disadvantage in Poketree).

This structure is $\mathcal{O}(\lg \lg n)$-competitive and it has a $\mathcal{O}(\lg n)$ worst-case cost because in the worst-case one additional operation is done in each node (trying the dynamic pointer). Moreover, we can support the update operations, although these can be quite complicated.

4.7.4 Discussion on the Poketree

One viewpoint to this approach and other similar algorithms is that we want to take advantage of structure in the input. In Poketree we augment a standard search structure with local caching in each node, in contrast to other competitive algorithms that actually change the structure of the whole search tree.

One advantage in Poketree is its flexibility: we can control the amount of work we do, because we can either restrict the number of dynamic pointers or we can place more items to a single node. The latter approach results

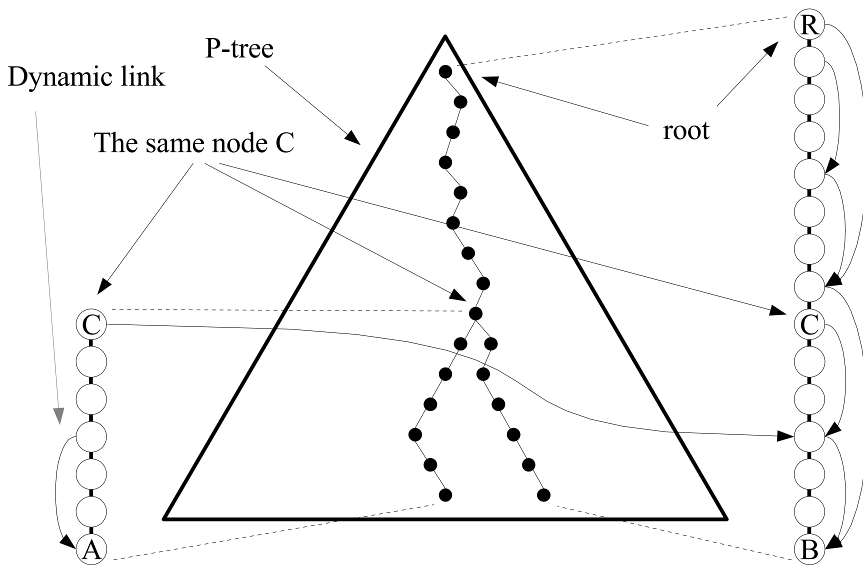

Figure 4.3: An example of a Poketree, where we show two root-to-leaf paths after first searching for the node A and then for the node B. The nodes A and B share a common ancestor C and the paths are shown at the left and right side of the tree. The dynamic pointers are drawn as arrows that point from one node to another.

	Tango	MST	Poketree(RB)
Search, worst-case	$\mathcal{O}(\lg\lg n \lg n)$	$\mathcal{O}(\lg^2 n)$	$\mathcal{O}(\lg n)$
Search, amortized	$\mathcal{O}(\lg\lg n \lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$
insert/delete	not supported	$\mathcal{O}(\lg^2 n)$	$\mathcal{O}(\lg n)$
memory per node	4w+2lg w+2	7w+lg w+1	6w+2

Table 4.4: Known asymptotic upper bounds on the performance of competitive BST algorithms. Memory is given in bits, where letter w is a shorthand for "words", and we assume that only one item is stored at a node. The MST algorithm is the multisplay tree [72] and Poketree(RB) is Poketree based on a RB-tree.

in a B-tree, where the cost incurred by dynamic pointers per node remains a constant, but the cost per item drops down.

Another advantage is a low worst-case cost. For example, the worst-case cost of RB-trees is very good, in theory the tree height is $2\lg n$ and in practice for uniform data the average access depth is between $(\lg n - 1)$ and $\lg n$ [1] which is the best possible. For a comparison of asymptotic costs of different algorithms see Table 4.4. We also observe that the updates in Poketree are more natural than in other approaches, because in the multisplay tree the P-tree is stored in the nodes, but is only used during updates.

It remains an open problem to study how these algorithms perform in empirical experiments. Even on structured data these algorithms pay $\mathcal{O}(\lg\lg n)$ per operation and in the worst-case they could be even slower than a simple balanced BST.

Chapter 5

Computing I/O Efficiently an Approximate BST

In the previous chapter we were interested in online BST algorithms and one of them, BST-FPL, required an algorithm to compute a nearly optimal BST for items when we have known weights or probabilities for the items. In this chapter we study how to solve this problem efficiently.

5.1 Background

We call the problem of computing an approximate BST as APPROX-BST. Formally it is: given items $\{1, \ldots, n\}$ and associated probabilities $\langle p_1, \ldots, p_n \rangle$, we want to compute a BST where an item i is found after searching d_i nodes, such that the expected path length $\sum_{i=1}^{n} p_i \, d_i$ is almost minimized. In practice this means that $d_i \leq -\lg p_i + c$ for some constant c, which implies that the expected cost is $H + c$, where $H = (\sum_{i=1}^{n} -p_i \lg p_i)$ is the entropy of the item distribution.

APPROX-BST is not limited to our interest only; it is a fundamental problem given attention by Knuth [48] and in the recent literature, for example, Brodal and Fagerberg [20] need approximately optimal BST tree, and so does Gagie [41], and Ailon et al. [2]. In all of these articles the authors design a data structure and desire bounds on the running time,

such as a good string dictionary with a low search time, and they need a nearly optimal BST to realize this.

Recall that the exact solution minimizing the expected path length due to Knuth is $\Theta(n^2)$ both in time and space. Because such a quadratic cost with data structures is prohibitively high in many applications, several authors have worked on approximation algorithms [48, 45, 36, 56, 75]. For n items in the BST the running time of these algorithms is typically either $\mathcal{O}(n)$ or $\mathcal{O}(n \lg n)$.

At first it might appear that these running times are optimal (ignoring the constant factors), because if n items are accessed then the cost of building a BST is $\Omega(n)$. Nevertheless, in the next section we discuss the reason why we were somewhat dissatisfied with the time complexities of the above algorithms, which in short is how the computational costs are counted.

5.2 Costs in Data Structures

The bounds $\mathcal{O}(n)$ and $\mathcal{O}(n \lg n)$ for the running times in the previous section are expressed in the unit cost model in which each performed *instruction* is given a cost of one. This model was given attention in Section 4.2 on page 26, where we discussed the cost of the searches in BSTs and other related structures. The unit-cost model is computation, not data, centric, but nevertheless the CPU that performs the instructions is not necessarily the bottle-neck in the computation. Rather, the cost of accessing the data in memory (any primary data storage like RAM, hard disk, or tape) may be the most significant factor in the run-time, as van der Pas [67] notes in his introduction to caching. This factor has naturally been recognized for a long time, see for example the following quote from Backus' 1977 Turing lecture [13]:

> "Von Neumann computers are built around a bottleneck: the word-at-a-time tube connecting the CPU and the store."

The importance of data transfers implies that we should at least consider taking into account the I/O -issues when modeling the costs. The simplest

method is to count the number of (distinct) memory accesses of operations, as is done for example in the cell probe model [57], where also the word length is limited to $\lg n$ bits for a data structure storing n things. However, these models that count the accesses ignore the fact that computers may store data to hardware such as the cache, standard memory, hard drives, and tape drives, and these have costs in different orders of magnitude. Also, the significance of the difference in latencies has grown as the following quote from [49] demonstrates:

> "The time to service a cache miss to memory has grown from 6 cycles for the Vax 11/780 to 120 for the AlphaServer 8400."

The external memory model(s) [70] distinguish between a fast work memory and a slower external memory. The external memory is divided into pages of B units and we pay a cost of one for fetching a page to the work memory. The objective in this model is to minimize these page fetches. Memory-disk interactions is an application, for example we can analyze B-trees[1] in this setting [48]. We can also analyze interactions between a cache on the CPU and the main memory, but then the read and write-operations must behave similarly in cost, i.e., the cache has to be write-back. The algorithms for the external memory model are called *cache-aware*, because they can rely on knowing the value B, which is a property of the cache and may thus differ between platforms.

Knowing B, though, is not a requirement for performing well cache-wise, because there are algorithms like the quicksort that empirically perform well if cache misses are measured [49] though they are not cache-aware. This is because we can use operations, such as linear scans, which cause few cache misses for all values of B. For more complicated operations, such as storing a matrix for multiplication, we need more complicated constructions, such as the ones used by Veldhuizen [69].

A formal framework was given by Frigo et al. [39] who coined the term cache-oblivious algorithm. In this framework we design an algorithm which

[1] The B-tree is a name of the algorithm and the "B" in it is not to our knowledge referring to the same B as the cache line length.

is oblivious to the value B. An optimal cache-oblivious algorithm causes $\mathcal{O}(M^*)$ cache misses, where M^* is the number of cache misses made by any cache-aware algorithm. Besides giving several cache-oblivious algorithms, they showed the following result.

Theorem 5. *Design a cache-oblivious algorithm for two-levels of memory hierarchy, and in the design assume that:*

1. *The cache is tall; i.e., the size of the cache is $\Omega(B^2)$.*
2. *The page replacement policy is optimal; i.e., replace the page which is used farthest in the future [14].*
3. *The cache is fully associative, i.e., each page can be placed to any position in the cache [44].*

Then an optimal cache-oblivious algorithm designed for two levels of memory causes asymptotically optimal number of cache misses for multiple levels of hierarchy. That is, if at a certain level of a hierarchy a cache-aware algorithm causes M^* cache misses, then the cache-oblivious algorithm causes $\mathcal{O}(M^*)$ cache misses. The memory hierarchy must have the property of inclusiveness*, which means that if an item is stored in a cache at a level i of the hierarchy, then it is also stored at the larger and slower cache at a level $i+1$.*

Note that Frigo et al. show that none of the unrealistic design assumptions need to be true, for example the LRU page replacement policy works almost as well as the optimal one. So the point is that cache-obliviousness of an algorithm implies a good performance for several layers of memory.

5.3 Our Algorithm: AWOBST

Now we have the importance of I/O in mind and, hence, we want an algorithm that constructs a BST with few I/O-operations. Actually, there is already an algorithm for this problem in the cache-oblivious model: Brodal

	instructions	space	cache misses	output
Optimal Knuth [48]	$\Theta(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	best
Approximation Knuth [48]	$\mathcal{O}(n \lg n)$	$\Theta(n)$	$\mathcal{O}(n \lg n)$	$H+1$
Brodal and Fagerberg [20]	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(n/B)$	$2\,H+2$
AWOBST [P5]	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(n/B)$	$H+1$

Table 5.1: Certain algorithms for constructing weighted BSTs, their instruction count, space usage, number of cache misses in the cache-oblivious model and how good is the expected performance of the constructed BST.

and Fagerberg [20] give an algorithm with a cost of $\mathcal{O}(n/B)$, but unfortunately the quality of the produced BST is not close to the optimal. As stated above, the previous upper bound for the average cost was $H + 1$, but Brodal and Fagerberg's solution produces trees with a bound $2H + 2$. What we want is the best of both worlds: I/O performance *and* a BST with performance scaling in H, not $2H$. Publication [P5] shows that this is possible. See Table 5.1 for a rough comparison of construction algorithms for BSTs.

Publication [P5] also achieves a theoretical upper bound for the cost of the constructed BST that is slightly better than the previous ones and this bound implies the following new upper bound for the expected performance of the optimal BST:

$$H + 0.087 + \lg(1 + p_{\max}),$$

where $p_{\max}$ is the maximal probability on items. The previous best was $H + 1$. Also, if $p_{\max} < 2/3$ we obtain a bound $H + 0.503$.

5.3.1 Description of AWOBST

The name of the algorithm is arithmetic weight-optimal binary search tree (AWOBST). It is conceptually simple: assign to each item i a priority that is the index of the first bit that changes between the binary representations of $\sum_{j=1}^{i-1} p_j$ and $\sum_{j=1}^{i} p_j$ (this priority equals the length of a code word assigned to the item when encoded with the arithmetic code [63]). Then

build a BST in a heap-order with respect to these priorities. This building process can be visualized as doing a so-called *postorder* traversal of the BST while reading the sorted items from an array. Recall that in the postorder visit of a BST we process the items in the following order: first process the left child, then the right child, and finally the node itself. We can implement the construction process with a scan and a stack, because we can store the parent and left sub-tree in the stack while constructing the right sub-tree. We emphasize that we do not use any non-realistic bit tricks, though we rely on the binary presentations of the probabilities.

The number of cache misses with AWOBST is $\Theta(n/B)$, and the implementation is independent of the value B, thus it is a cache-oblivious algorithm. We think that it is not possible to save more than a constant amount cache misses even if we knew B, hence using the cache-oblivious model does not result in a overhead of $\Omega(n/B)$ in this case, although this depends on how we want to place the constructed BST to the memory.

We emphasize that it is the process of constructing the BST which is cache-oblivious in AWOBST, and in Publication [P5] we do not consider the performance of the constructed BST, because this is a separate problem. For completeness, in Section 5.4 we give details on how to make the produced BST perform well in the cache-oblivious model.

5.3.2 An example

We give an example on how a certain BST is constructed in AWOBST. The following Table 5.2 gives the input.

item i	1	2	3	4	5	6
probability p_i	0.001	0.001	0.011	0.0001	0.0001	0.01

Table 5.2: An example of an input to AWOBST.

From the input we calculate the cumulative probabilities in binary, and hence also the priorities which are the indices of the first changing bit between two successive cumulants. We give these in Table 5.3.

item i	1	2	3	4	5	6
$\sum_{j=1}^{i} p_j$	0.001	0.01	0.101	0.1011	0.11	0.111...
d_i	3	2	1	4	2	3

Table 5.3: Cumulative probabilities and the priorities of the items in AWOBST, calculated from the input in Table 5.2.

From the priorities we can deduce the tree in Figure 5.1, because from the priorities we can infer the child-parent relationships.

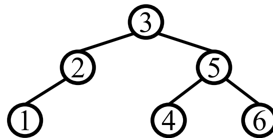

Figure 5.1: The BST that AWOBST constructs from the input given in Table 5.2.

As we stated earlier, a stack is used to store a sub-tree consisting of a node and its left descendants while constructing the sub-tree consisting of right descendants. Hence, first we process item 1, push it to stack, then process item 2 and also push it to the stack. Then we process item 3 and because its priority is smaller than either the priority of item 1 or 2, we know that all items in the sub-tree rooted at item 1 or 2 have been found. Hence we can pop items 1 and 2 from the stack, link them together with item 3 and push it to the stack. We continue and process the right sub-tree of item 3, consisting of items 4, 5, and 6, and after this we note that all items have been processed and we link the remaining items in the stack and then we have the final tree.

5.4 I/O performance of the constructed BST

Publication [P5] does not bound the performance of the constructed BST in the cache-oblivious model. Rather it bounds the expected path length of an access. Recall that this does not necessarily model the performance of a BST well, because for example a parent and a child node are more likely to be accessed together. The problem of placing the items to memory is a separate problem from the BST construction. We now give details on the memory layout, because this is not completely trivial and otherwise an argument could be raised that we do not use cost models consistently, but pick the model which best fits our analysis.

5.4.1 Where to place the items?

Brodal et al. [21, 22] experimented with several memory layouts for BSTs and for us the interesting ones are an “ad-hoc” B-tree layout where we just guess some value for the page size B and a cache-oblivious layout. The B-tree layout is interesting for its simplicity and empirical performance, and the cache-oblivious layout for its theoretical properties.

For perfectly balanced BSTs the layouts are as follows. In the B-tree layout we first place together the topmost B items to the memory and then recurse to the remaining sub-trees until we have no items.

The cache-oblivious layout is more complicated, and was suggested by Prokop [61] and is called the van Emde Boas structure for its original inventor in a different context [68]. Let $R(T)$ denote the function that for a balanced BST T outputs the memory layout for T. Let h be the height of T and let the sub-tree T_0 consist of the topmost half of T and let sub-trees from T_1 to $T_{2^{h/2}}$ be the ones that remain at the bottom half of the T. The van Emde Boas layout is then given by the following recursive formula:

$$R(T) = R(T_0) \textbf{ catenate } R(T_1) \textbf{ catenate } \ldots \textbf{ catenate } R(T_{2^{h/2}}).$$

In theory the van Emde Boas structure is not as efficient as the B-tree layout, if the node size of the B-tree matches the page size B. Then an access to a depth d costs $d/\lg B + 1$ misses in the B-tree layout, but the

cost is upper bounded only with $4d/\lg B + 1$ in van Emde Boas layout. We remind that the van Emde Boas structure is still asymptotically optimal for unknown B and for multiple levels of memory hierarchy.

These bounds $\mathcal{O}(d/\lg B)$ are for a balanced tree, but the bounds also apply to BSTs with other shapes if the "missing items" are simply ignored by removing the gaps in the memory where they would have been. Hence, because the item i is at most at the depth $\lg p_i + 1$ we can achieve $\mathcal{O}(-\lg_B p_i)$ cache misses with the I/O efficient layouts.

5.4.2 How to place the items?

The details in this section are somewhat technical. We describe how to assemble the layouts discussed in the previous section with AWOBST. Actually we can apply the following method for any layout where we can efficiently obtain an order-preserving (as defined later) position of a node in the final BST. We apply the standard time forward processing technique [25]. In short, we delay computation depending on data that is not in the cache to the future using a cache efficient heap (for example in the cache-oblivious model this heap is naturally a cache-oblivious heap).

The idea is as follows. When the parent of a node is set then we know the children of this node. If we knew the structure of the BST, then we could calculate the memory location for this node (and the locations of, or pointers to, its children) and push the node to a cache efficient heap using a priority that is the location. After AWOBST finishes, we would then pop all the items in the order of memory location and place them, and we would be finished.

The only significant "but" remaining is that when the parent of the node is set, we do not have enough information to infer the location of the node. This is because parts of the BST have not been constructed. Hence instead of the real location we use an order-preserving location, which guarantees that the priority does not need to equal the actual location. Instead, the order relation between the priorities must be conserved, i.e., nodes in higher memory locations have also larger priorities. Also, as we need to know the pointers to the children, we need to do an additional trick with heaps, as

described in Table 5.4.

We did not describe how to compute the order-preserving location for the nodes. For the item i we can do this with the following information:

- Probability p_i of the item i.
- Cumulative probability $\sum_{j=1}^{i} p_j$.
- Minimal probability $p_{\min}$ over all items.

The idea is to embed the constructed BST into a perfectly balanced BST of height $\lg 1/\lfloor\lfloor p_{\min}\rfloor\rfloor$, where $\lfloor\lfloor x\rfloor\rfloor$ denotes the closest power of two that is smaller than x. See for example Figure 5.2 where we have embedded the BST from Figure 5.1. Note that the depth and position of each item is inferred directly from the probabilities so there is a gap between the item 5 and 4.

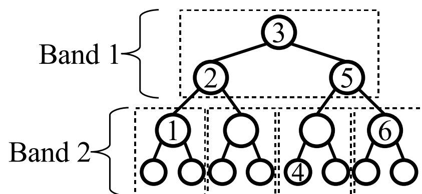

Figure 5.2: How a BST from Figure 5.1 is embedded to a larger tree. Here B is 3 and the boxes correspond to nodes in the B-tree which stores three items per node.

Now the order-preserving location is easy to calculate for the B-tree layout. Given the item i we can obtain the actual position in the embedded tree from the probability p_i and the cumulative probability (the position is at the depth of the priority and there are as many items to the left of the item as how many times the priority bit can have changed in the cumulative probability). Then we just calculate the "box" where the item falls by calculating the band where the item is and how many boxes there are above and left to the position of the item i. This takes $\mathcal{O}(1)$-time.

Input: A cache efficient heap H_a which contains the nodes with order-preserving priorities. The nodes include the keys the of children, but no pointers.

1. Set $S := 1$.
2. Until H_a is empty:
   (a) Pop a node N from H_a.
   (b) Assign N a memory location S.
   (c) $S := S + 1$.
   (d) Push N to a heap H_b with a priority equal to its key.
   (e) Push the memory location and the key of N to the heap H_b two times, with the priority equal to the key of the left and with the priority equal to the key of the right child.
3. Until H_b is empty:
   (a) Pop a node N and the memory locations of its children from H_b.
   (b) Push N, with correct child pointers, to a heap H_c with priority equal to the memory location of N.
4. Pop the nodes from H_c and assign them to the correct memory locations.

Table 5.4: How to I/O efficiently place the nodes to the memory, a general approach.

The process is similar for the van Emde Boas layout. However, to our knowledge one must use a recursive function with possibly $\mathcal{O}(\lg 1/\lfloor\lfloor p_{\min}\rfloor\rfloor)$ levels of recursion.

Chapter 6

Conclusions

In this Thesis, we have studied online problems in which, in general, we learn good decisions from the past data. Our approach is theoretical and we provided formal performance guarantees. First we studied the FPL algorithm, and we showed that we can effectively apply it to the bandit setting. This results in an approach that is more flexible than the previous algorithms, but suffers from increased complexity in estimating the probability of selecting an expert.

In later chapters we were interested in understanding properties of BST algorithms and we mostly used methods developed for the online setting to derive results. We showed that if we generate accesses to a BST with a random source, then all BST algorithms have a cost proportional to the randomness in this source, as measured by the entropy. Also, the cost is lower bounded by the complexity of the access sequence, as measured by the Kolmogorov complexity. It remains an open problem to effectively compute a strict lower bound to the cost of the best BST algorithm for any given access sequence. Lower bounds are interesting because they give not only a limit to our performance, but we can also potentially use them to design better algorithms.

One such application of lower bounds is in our study on augmenting a tree structure, such as a B-tree, with dynamic pointers that change during the accesses. The advantage in the additional dynamic pointers is that they

capture the regularities in the input. Formally our construction achieves a cost of $\mathcal{O}(\lg \lg n)$ times the cost of any BST algorithm. However, we still need to study when this construction (or other similar algorithms) results in increased performance in empirical experiments.

Finally, we gave an algorithm which is I/O efficient and computes an approximately optimal BST for any given weights on items. This work might also have a minor importance in the coding theory, as we provide new upper bounds to the code length of a one-to-one code. Our upper bound depends on the maximum probability on the items.

There are essentially two factors (or open problems) which have enabled this work. The first is the fact that we did not and still do not understand very well how to deal with uncertainty (or regularity) in the future inputs. The second is that computers are human made objects and are evolving. What holds today, does not necessarily hold in the future, as we can see from the relative growth of importance of I/O and parallelism.

Bibliography

[1] Timo Aho, Tapio Elomaa, and Jussi Kujala. Reducing splaying by taking advantage of working sets. Accepted to WEA, 2008.

[2] Nir Ailon, Bernard Chazelle, Seshadhri Comandur, and Ding Liu. Self-improving algorithms. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 261–270, New York, NY, 2006. ACM press.

[3] Susanne Albers. Online algorithms: A survey. *Mathematical Programming*, 97(1-2):3–26, 2003.

[4] Susanne Albers and Marek Karpinski. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.

[5] Susanne Albers and Stefano Leonardi. On-line algorithms. *ACM Computing Surveys*, 31(3es), 1999. Article number 4.

[6] Noga Alon and Alon Orlitsky. A lower bound on the expected length of one-to-one codes. *IEEE Transactions on Information Theory*, 40(5):1670–1672, 1994.

[7] Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. Fusion trees can be implemented with AC0 instructions. *Theoretical Computer Science*, 215:337–344, 1999.

[8] Spyros Angelopoulos, Reza Dorrigiv, and Alejandro López-Ortiz. On the separation and equivalence of paging strategies. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 229–237, Philadelphia, PA, USA, 2007. SIAM.

[9] Peter Auer. Using upper confidence bounds for online learning. In *Proceedings of the Fourty-First Annual Symposium on Foundations of Computer Science*, pages 270–279, Los Alamitos, CA, 2000. IEEE Computer Society Press.

[10] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The non-stochastic multi-armed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[11] Baruch Awerbuch, David Holmer, Herb Rubens, and Robert Kleinberg. Provably competitive adaptive routing. In *INFOCOM: Twenty-Fourth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 631–641 vol. 1. IEEE Computer Society Press, 2005.

[12] Baruch Awerbuch and Robert Kleinberg. Near-optimal adaptive routing: Shortest paths and geometric generalizations. In *Proceeding of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, pages 45–53, New York, NY, 2004. ACM Press.

[13] John W. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[14] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

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

[16] Avrim Blum. On-line algorithms in machine learning. In *Developments from a June 1996 seminar on Online algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 306–325, London, UK, 1998. Springer-Verlag.

[17] Avrim Blum, Carl Burch, and Adam Kalai. Finely-competitive paging. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 450, Washington, DC, USA, 1999. IEEE Computer Society.

[18] Avrim Blum and Yishay Mansour. Learning, regret minimization, and equilibria. In Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani, editors, *Algorithmic Game Theory*, pages 79–102. Cambridge University Press, New York, NY, USA, 2007.

[19] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[20] Gerth Stølting Brodal and Rolf Fagerberg. Cache-oblivious string dictionaries. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 581–590, New York, NY, USA, 2006. ACM Press.

[21] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, Philadelphia, PA, USA, 2002. SIAM.

[22] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache-oblivious search trees via trees of small height. Technical Report ALCOMFT-TR-02-53, ALCOM-FT, May 2002.

[23] Nicolò Cesa-Bianchi, Yoav Freund, David P. Helmbold, David Haussler, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. *Journal of the ACM*, 44(3):427–485, 1997.

[24] Nicolò Cesa-Bianchi and Gábor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, Cambridge, UK, 2006.

[25] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Je ffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. SIAM.

[26] Richard Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[27] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting $\log n$-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, Boston, USA, second edition, 2001.

[29] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, USA, 1991.

[30] Varsha Dani and Thomas P. Hayes. How to beat the adaptive multi-armed bandit. Technical report, Cornell University, 2006. http://arxiv.org/cs.DS/602053.

[31] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–14, New York, NY, USA, 1997. ACM.

[32] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality – almost. In *Proceedings of the Fourty-Fifth Annual IEEE Symposium on Foundations of Computer Science*, pages 484–490, Washington, DC, 2004. IEEE Computer Society Press.

[33] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[34] Reza Dorrigiv and Alejandro López-Ortiz. A survey of performance measures for on-line algorithms. *SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 36(3):67–81, 2005.

[35] Reza Dorrigiv and Alejandro López-Ortiz. Closing the gap between theory and practice: New measures for on-line algorithm analysis. In *WALCOM: Algorithms and Computation*, volume 4921 of *Lecture Notes in Computer Science*, pages 13–24, 2008.

[36] Michael L. Fredman. Two applications of a probabilistic search technique: Sorting X+Y and building balanced search trees. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, pages 240–244, New York, NY, USA, 1975. ACM Press.

[37] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[38] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, volume 904 of *Lecture Notes in Computer Science*, pages 23–37, London, UK, 1995. Springer-Verlag.

[39] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the Fourtienth Annual Symposium on Foundations of Computer Science*, pages 285–297, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[40] Martin Fürer. Randomized splay trees. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 903–904, Philadelphia, PA, 1999.

[41] Travis Gagie. Restructuring binary search trees revisited. *Information Processing Letters*, 95(3):418–421, 2005.

[42] James Hannan. Approximation to Bayes risk in repeated plays. In M. Dresher, A. Tucker, and P. Wolfe, editors, *Contributions to the Theory of Games*, volume 3, pages 97–139. Princeton University Press, Princeton, NJ, 1957.

[43] Dion Harmon. *New Bounds on Optimal Binary Search Trees.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2006.

[44] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[45] T.C. Hu and A.C. Tucker. Optimum computer search trees and variable-length alphabetical codes. *SIAM Journal Applied Mathematics*, 21(4):514–532, 1971.

[46] Sham M. Kakade, Adam Tauman Kalai, and Katrina Ligett. Playing games with approximation algorithms. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, pages 546–555, New York, NY, 2007. ACM Press.

[47] Adam Tauman Kalai and Santosh Vempala. Efficient algorithms for online decision problems. *Journal of Computer and System Sciences*, 71(3):26–40, 2005.

[48] Donald Ervin Knuth. *The Art of Computer Programming, 2nd Ed. (Addison-Wesley Series in Computer Science and Information.* Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1998.

[49] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, Philadelphia, PA, USA, 1997. SIAM.

[50] John Langford and Tong Zhang. The epoch-greedy algorithm for multi-armed bandits with side information. In *Neural Information Processing Systems*, 2007.

[51] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer Verlag, second edition, 1997.

[52] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. In *IEEE Symposium on Foundations of Computer Science*, pages 256–261, 1989.

[53] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, 1994.

[54] Joan Marie Lucas. Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Computer Science Department, Rutgers University, 1988.

[55] H. Brendan McMahan and Avrim Blum. Geometric optimization in the bandit setting against an adaptive adversary. In J. Shawe-Taylor and Y. Singer, editors, *Proceeding of the Seventeenth Annual Conference on Learning Theory*, volume 3120 of *Lecture Notes in Computer Science*, pages 109–123, Berlin, Heidelberg, 2004. Springer.

[56] Kurt Mehlhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6(2):235–239, 1977.

[57] Peter Bro Miltersen. Cell probe complexity – a survey. In *Pre-Conference Workshop on Advances in Data Structures at the Nineteenth Conference on Foundations of Software Technology and Theoretical Computer Science*, 1999.

[58] Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 367–375. SIAM, 1992.

[59] Neural information processing systems 2007 workshop on machine learning in adversarial environments for computer security. http://mls-nips07.first.fraunhofer.de/.

[60] Ben Pfaff. Performance analysis of BSTs in system software. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 410–411. ACM Press, 2004.

[61] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1999.

[62] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[63] Jorma Rissanen and Glen G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23:149–162, 1979.

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

[65] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[66] Robert Endre Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 4(5):367–378, 1985.

[67] Ruud van der Pas. Memory hierarchy in cache-based systems. Technical Report 817-0742-10, Sun microsystems, 2002.

[68] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory (Theory of Computing Systems)*, 10(1):99–127, 1977.

[69] Todd L. Veldhuizen. Scientific computing: C++ versus Fortran: C++ has more than caught up. *Dr. Dobb's Journal of Software Tools*, 22(11):34, 36–38, 91, November 1997.

[70] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[71] Volodimir G. Vovk. Aggregating strategies. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, pages 371–386, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[72] Chengwen Chris Wang, Jonathan Derryberry, and Daniel Dominic Sleator. O(log log n)-competitive dynamic binary search trees. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 374–383. ACM Press, 2006.

[73] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989.

[74] Hugh E. Williams, Justin Zobel, and Steffen Heinz. Self-adjusting trees in practice for large text collections. *Software–Practice & Experience*, 31(10):925–939, 2001.

[75] Raymond Yeung. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, 37(3):564–572, 1991.

[76] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In T. Fawcett and N. Mishra, editors, *Proceeding of the Twentieth International Conference on Machine Learning*, pages 928–936, Menlo Park, CA, 2003. AAAI Press.